TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

DAVOOD RASTI
ALIREZA RASTI

AUGMENTED REALITY FRAMEWORK AND DEMONSTRATOR

Master of Science Thesis

Examiners: Adjunct Prof. Pekka Jääskeläinen and
Timo Viitanen, MSc
Examiner and topic approved on 1st of March 2017

ABSTRACT

DAVOOD RASTI & ALIREZA RASTI: Augmented Reality Framework and Demonstrator
Tampere University of technology
Master of Science Thesis, 86 pages
May 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Adjunct Prof. Pekka Jääskeläinen and Timo Viitanen, MSc

Keywords: Augmented Reality, Framework, Mobile Augmented Reality, Rendering.

Augmenting the real-world with digital information can improve the human perception in many ways. In recent years, a large amount of research has been conducted in the field of *Augmented Reality* (AR) and related technologies. Subsequently, different AR systems have been developed for the use in different areas such as medical, education, military, and entertainment.

This thesis investigates augmented reality systems and challenges of realistic rendering in AR environment. Besides, an object-oriented framework, named ThirdEye, has been designed and implemented in order to facilitate the process of developing augmented reality applications for experimental purposes. This framework has been developed in two versions for desktop and mobile platforms. With ThirdEye, it is easier to port the same AR demo application to both platforms, manage and modify all AR demo application components, compared to the various existing libraries. Each feature that the ThirdEye framework includes, may be provided by other existing libraries separately but this framework provides those features in an easy-to-use manner.

In order to evaluate usability and performance of ThirdEye and also for demonstrating challenges of simulating some of the light effects in the AR environment, such as shadow and refraction, several AR demos were developed using this framework. Performance of the implemented AR demos were benchmarked and bottlenecks of different components of the framework were investigated.

This thesis explains the structure of the ThirdEye framework, its main components and the employed technologies and the *Software Development Kits* (SDKs). Furthermore, by using a simple demo, it is explained how this framework can be utilized to develop an AR application step by step. Lastly, several ideas for future development are described.

PREFACE

I would like to express my gratitude to my supervisors, Pekka Jääskeläinen and Timo Viitanen for their supports, guidance and advices during this thesis. In addition, thanks to my brother and my partner in the thesis Alireza Rasti for his help and efforts. Finally, I would like to thank my lovely parents and sister who encouraged me throughout the thesis.

Tampere, 22.05.2018

Davood Rasti

I would like to thank my supervisors Pekka Jääskeläinen and Timo Viitanen for their guidance, help and comments and for suggesting the interesting topic for this thesis. Also, I would like to thank my best friend, best mate and best brother Davood Rasti, who is my partner in this thesis. I really appreciate him for his patience, support and cooperation during our Bachelor and Master study. My very special thanks go to my parents and my lovely sister for their infinite support spiritually and financially, especially for these past ten years.

Tampere, 22.05.2018

Alireza Rasti

DIVISION OF WORK

This thesis has been done by Davood Rasti and Alireza Rasti (Rasti brothers). The research phase and designing of the framework and the demo application have been done conjointly by both members. The implementation of the thesis work was divided equally based on the workload of the tasks. The contribution of each member in writing the thesis is presented in the table below.

Chapters	Sections	Author
1. INTRODUCTION	1. Introduction	Davood & Alireza
2. AUGMENTED REALITY	2.1 Augmented Reality Systems	Davood
	2.2 Structure	Alireza
	2.3 Pose Tracking Methods	Davood & Alireza
	2.4 Augmented Reality Devices	Alireza
3. OBJECT RENDERING IN AR	3.1 Rendering	Davood
	3.2 Rasterization	Alireza
	3.3 Ray Tracing	Alireza
	3.4 Shadow	Alireza
	3.5 Transparency and reflection	Alireza
	3.6 Photorealistic Rendering	Davood & Alireza
	3.7 Caustics Implementation	Alireza
	3.8 Light Direction Estimation	Davood
4. RELATED WORK	4.1 ARToolkit	Davood
	4.2 ARToolkitPlus	Davood & Alireza
	4.3 ArUco	Davood
	4.4 Google ARCore	Alireza
	4.5 Apple ARKit	Alireza
5. THIRDEYE AR FRAMEWORK	5.1 Overview	Davood & Alireza
	5.2 ThirdEye – Desktop	Davood & Alireza
	5.3 ThirdEye – Mobile	Davood
6. EVALUATION	6.1 Desktop AR Applications	Davood
	6.2 Mobile AR Applications	Davood
	6.3 Benchmarking	Davood & Alireza
7. FUTURE WORK	7.1 Future Work	Davood
8. CONCLUSIONS	8. Conclusions	Davood & Alireza

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

2D	Two-Dimensional
3D	Three-Dimensional
API	Application Programming Interface
AR	Augmented Reality
CPU	Central Processing Unit
FPS	Frames Per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
HMD	Head Mounted Display
IMU	Inertial Measurement Unit
MR	Mixed Reality
OS	Operating Systems
SDK	Software Development Kit
UI	User Interface
VR	Virtual Reality

1. INTRODUCTION

Nowadays, many of our professional and everyday tasks rely on digital information. We can almost find useful information about everything that we need using technologies and gadgets. Usually information is presented on a device's display and it is difficult to associate it with the real world. For example, navigation directions on a mobile phone is just displayed on the phone's screen and it is not easy to match the direction with the real world. Combining computer-generated information with our surrounding physical environment can improve our perception in different ways and simplify our jobs and daily tasks.

Augmented Reality (AR) is a technology which mixes a real-world environment with computer-generated information in real-time or non-real-time for different purposes. The generated information that is used in AR can be video, audio, text, even smell and touch sensations, or a combination of them that can enhance real word experiences around the users [1], [2]. Generally, AR systems consist of three main components: input, AR engine, and output, where the AR engine is responsible for tracking and augmented content. The types of components are dependent on the characteristics and objectives of the AR system, such as the type of input data, required accuracy of the tracking, type of augmented content, degree of realism of the generated content, and the target platform. For example, in the "Pokemon Go" game, the GPS data is the input data which is used for tracking, its type of augmentation is 3D visual content generated by rasterization rendering method and the target platform is mobile devices such as smartphones and tablets.

Nowadays, the main focus of augmented reality research is on visual AR, which is the most common type of augmented reality. In research and development of AR applications, different toolkits may be used in each part of the applications. Generally, academic studies concentrate on a specific component of the AR system. Hence, a framework can be used to facilitate the AR application development and prevent the redundancy by managing the components and handling the common parts.

The topic for this thesis was given by the *Virtual reality and Graphics Architectures* (VGA) group in *Tampere University of Technology* (TUT). The general objectives of this thesis were:

- Research augmented reality systems.
- Review the available open source tools for implementing the AR demo.
- Design and implement an AR framework.
- Implement AR demos using the framework.
- Implement light effects in AR environments using rasterization.

In this thesis the ThirdEye framework was designed and implemented based on the requirements and conducted research. This framework has been developed for desktop (Linux) and Android mobile platforms. The ThirdEye framework integrates main components that are required for AR application development. Both versions of the framework employ ARToolkitPlus for pose estimation which is an open-source and cross-platform tracking SDK (See Section 4.2). In the desktop version, OpenGL 3.3 and in the mobile version OpenGL ES 2.0 library are utilized for rasterization rendering. In order to accelerate the development process, a configuration management tool has been implemented. This tool helps to configure different parts of the framework. Compared to the various existing libraries, it is easy to port the same AR demo application to both platforms with ThirdEye. Furthermore, the ThirdEye framework has some features that facilitate developing augmented reality applications, especially for experimental purposes. Although each of these features may be provided separately by other existing libraries, ThirdEye provides them in an easy-to-use manner.

In order to evaluate the framework and demonstrate challenges of simulating physical light effects in an AR environment, various AR demos have been developed on both platforms. Some light effects, such as shadow and refraction effects in augmented environment, have been simulated in these AR demos. Furthermore, some of the demos were benchmarked and the results exposed the bottlenecks. Consequently, more benchmarks were done to investigate the bottlenecks of the framework. The benchmarks' results highlighted some important points that could be helpful for any AR application development.

This thesis is divided into the following chapters; Chapter 2 introduces augmented reality and AR systems; in addition, it discusses the structure of an AR system, and tracking methods in AR. Chapter 3 explains rendering techniques, light effects and related works in photorealistic rendering and light direction estimation in augmented reality. Chapter 4 discusses related works and a number of AR tools and libraries. Chapter 5 introduces the ThirdEye framework and describes its structure and features of both versions of the framework in detail; furthermore, a detailed guide is provided on how to use the ThirdEye framework. In Chapter 6, the implemented demos, evaluation of the framework and benchmarks are presented. Chapter 7 introduces several possible future work directions. Finally, Chapter 8 concludes the thesis.

2. AUGMENTED REALITY

The term of augmented reality was coined by Tom Caudell and David Mizell in 1992 for referring to the technology that they developed to help Boeing workers assemble aircrafts' electrical wires [3]. Paul Milgram and Fumio Kishino [4] consider AR as a part of a wider definition, *Mixed Reality* (MR). They defined Mixed Reality as an environment in which, *"real-world and virtual objects are presented together within a single display"*. All technologies that mix real world and digital information in any form are part of Mixed Reality. These combinations, based on how much real or virtual elements they have, can be categorized. Milgram and Fumio Kishino proposed *Reality-Virtuality (RV) Continuum* concept (Also known as Mixed Reality spectrum) (Figure 1) to connect pure real-world to the pure virtual world. The Mixed Reality term refers to the area between pure real environment to pure virtual environment this c*ontinuum*.

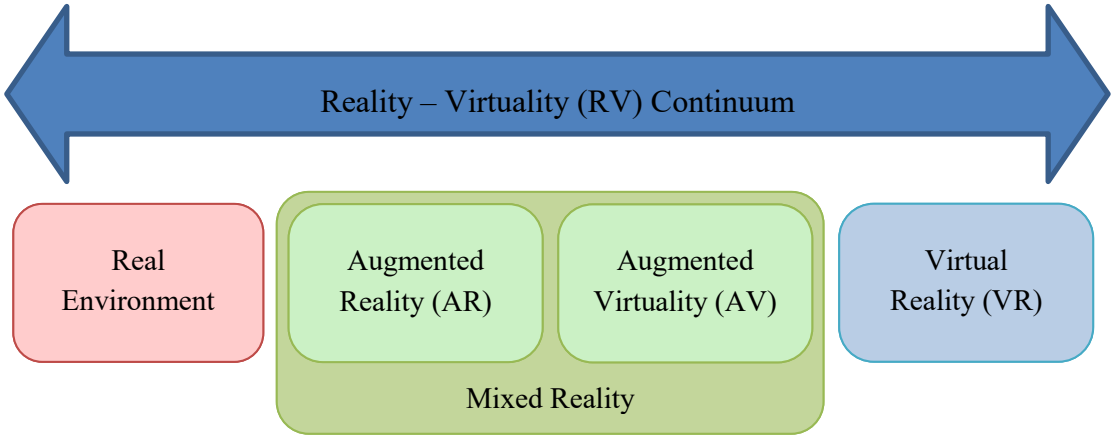


Figure 1. *Milgram's Reality – Virtuality (RV) continuum. (adapted from [4]).*

Real environment in reality-virtuality continuum, refers to real-world objects which can be observed directly or by conventional display [4]. In contrast, *virtual environment* refers to an environment which is completely computer-generated. Any other combinations of the real and virtual world, can be placed between these two, in the reality-virtuality Continuum.

Virtual Reality (VR) is a computer-generated environment in which participants completely immersed in, and are able to interact with it [4]. This environment can be totally fantasy or an accurate simulation of the real word physics. In general VR applications try to separate their users from real word and immerse them mentally (sometimes physically) in the digital environment. In order to do so, in VR, some of the participants' senses such

as sight, hearing and touch have been engaged, giving them feeling of being in the virtual environment.

Augmented Virtuality (AV) is a mainly virtual environment in which physical objects, people are added to it. In this system, usually multiple sensors, cameras and trackers are used to get required environmental information that are needed in the virtual world. AV-based system can be used for different purposes such training, entertainment and remote collaboration. For example, AV-based videoconference [5] in which participants from different locations are place behind a virtual table to simulate face-to-face meeting.

In contrast to the augmented virtuality and virtual reality that place their users in a mainly synthetic environment, augmented reality is the real world around the users which is augmented by computer-generated information. Usually augmented reality refers to a view of the real world that is overlaid by virtual geometry objects (visual augmented reality). In AR applications, the image of the real-world scene or other features of the environment around the user (like location, sound) are analyzed to generate augmented digital information.

Generally, augmented reality is considered as a system that adds digital information to the real world, although Azuma [2] did not limit AR to this functionality. In some AR applications it is needed to remove real object from real-world scene which is called *Diminished Reality (DR)*.

2.1 Augmented Reality Systems

Since the first augmented reality system was created by Ivan Sutherland in 1968 [6], much research has been done on this topic and related technologies such as computer vision, computer graphics and human interaction. As technology-leading companies around the world found AR promising, they have started competing each other to gain higher market share by investing in this area. In recent decade, as the result of these investments and research, AR systems have been evolved rapidly. Nowadays, there are many augmented reality systems which have been developed for different purposes, from entertainment and advertisement to training, engineering and military.

In general, we can categorize augmented reality systems in two groups, *portable* and *stationary* (Figure 2). Stationary AR systems like personal computers, video game consoles and projectors usually are equipped with powerful hardware. They can use more complicated computer vision algorithms to get better understanding from the real-world environment and provide high-quality augmented content. These kinds of system are more suitable for demanding processing task in AR application such photo-realistic rendering.

In contrast to the stationary systems, portable devices like mobile phones, tablets, AR helmets and smart glasses do not limit their users to specific location. The mobility of

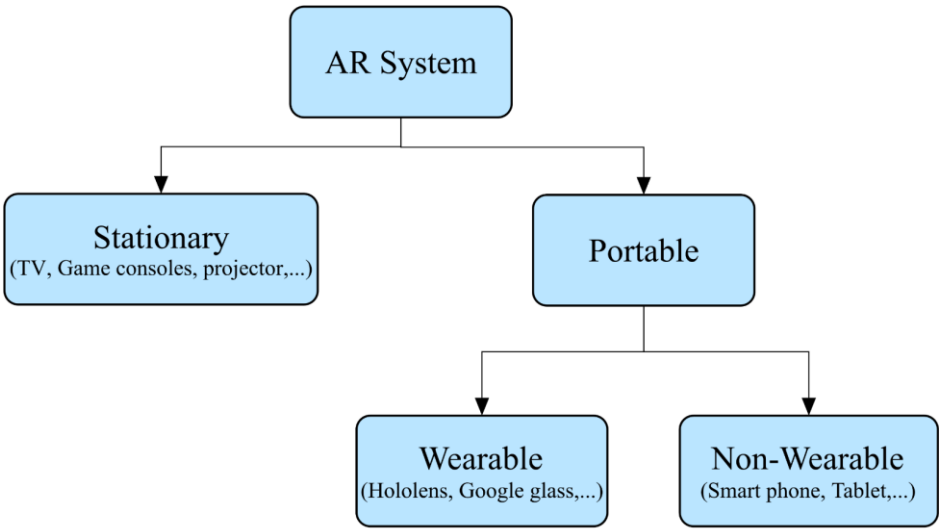


Figure 2. *AR System categories.*

these devices enables the use of augmented reality for a much wider range of purposes than stationary ones. Generally mobile augmented reality systems are integrated with different sensors such as *Global Positioning System* (GPS), *Inertial Measurement Unit* (IMU) and digital compass which are very useful for more accurate and robust tracking especially for *outdoor* environments. Mobile augmented reality systems can be classified as *wearable*, like smart glasses and AR helmets, and *non-wearable*, like smart phones and tablets. AR wearable devices give a better perception of the surrounding environment than non-wearable ones. They are hand-free which enable their users doing other tasks while receiving required information [7]. For example, workers of an assembly line can take advantage of assembly guidance provided by AR system, while doing their jobs. Usually, the users can interact with the system using voice commands or gesture or gaze. Although, wearable devices are not yet common as mobile phones and tablets, but there is huge potential for them in the future of the AR industry. *Google Glass* and *Microsoft HoloLens* are two examples of recent commercial AR wearable devices.

The portability of mobile devices comes at the cost of some hardware limitations. Portable devices usually have less processing power and memory, making them unsuitable for heavy tasks. *Distributed Augmented Reality* systems can be used to overcome this problem. In this approach, mobile devices capture required data and send it to a more powerful server, where the data is analyzed and the required augmented information is generated for processing. The generated content is then sent back to the mobile devices for visualization [8]. Figure 3 displays the architecture of distributed AR systems. In this system, mobile devices send data through the internet or wireless network. The quality of the network in distributed AR systems plays an important role. Data transmission latency has a big impact on the performance in these AR systems. With a high latency, result of syncing augmented content with real objects in the scene would be undesirable.

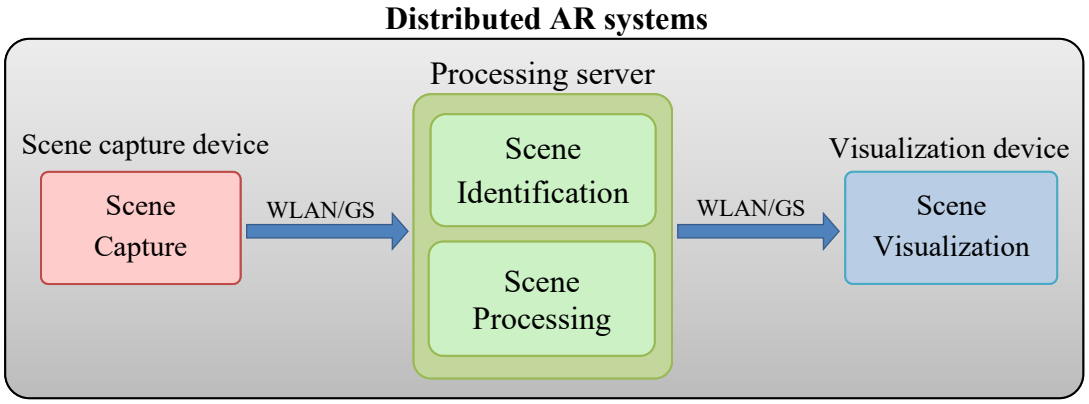


Figure 3. *Architecture of distributed AR systems (adapted from [8]).*

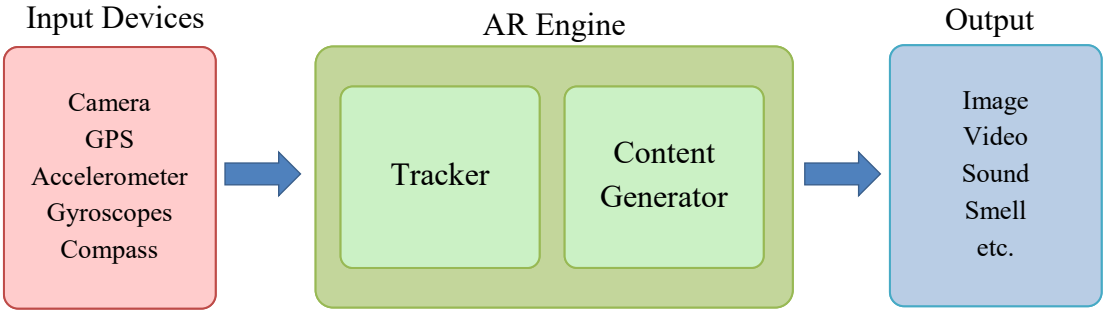


Figure 4. *Augmented Reality Structure.*

2.2 Structure

There are some fundamental and common tasks which each AR system should do, regardless of its type:

1) Tracks the real-world position and orientation of the device by processing surrounding physical environment data.
2) Creates necessary digital content (virtual objects, text, …), based on the system purpose.
3) Mixes the generated content and real environment.

Based on these tasks, we consider a basic architecture that all augmented reality systems can be based on. This architecture has three main components, which are *Tracking Devices, AR Engine* and *Output Device*. In this section these components and their functionalities are explained (Figure 4).

2.2.1 Input Devices

Every augmented reality system needs to have access to its surrounding physical world in order to analyze it and get required data. This can be achieved with different devices such as cameras (Color and RGB-D), GPS, IMU and mechanical tracking device, etc. Depending on the AR system's objectives, type and required tracking accuracy, one or combination of these devices can be used.

In following sections utilizing these devices in different augmented reality systems are discussed more.

2.2.2 AR Engine

In this architecture, main three augmented reality system tasks, that we have mentioned before, are done by AR Engine component. This component itself contains *Tracker* and *Content Generator* units. In following sections, we explain each of these units and their roles in an AR system.

- ***Tracker*:**
  In visual AR systems, augmented content should be aligned (registered) properly in the scene. In order to do so the virtual camera in the augmented environment should have same position and orientation as the real camera that captures real world scene. Estimating the parameters of the 6DOF (six degree of freedom) camera model by computing the position and orientation of the real camera is called pose estimation. There are different pose estimation methods that are useful for various situations. Choosing one of these methods for an AR system, depends on the system's design objectives, intended environment, and hardware constraints. Pose estimation plays an important role in the quality of augmented reality system. In some AR applications, such as medical and military uses, the robustness and accuracy of pose estimation is more crucial that others like games and entertainment. In these applications a small error may be unacceptable, so more sophisticated and accurate methods are used. Some augmented reality systems like portable ones that have limited hardware, should utilize lightweight methods that produce reasonable result with less calculation. There are several pose estimation methods which have been developed for different purposes. In general, we can classify these methods as *Vison-based, Sensor-based* or *Hybrid.* These methods are explained in Section 2.4.

- ***Content Generator*:**
  After pose estimation, the required virtual content is generated. In the AR engine, the content generator unit creates augmented materials based on the system's type and objective. In contrast to the complete virtual environment, in the augmented reality, it is usually required to render few virtual objects in the scene therefore,

Figure 5. *Compositing real-world image and computer-generated image.*

AR systems demand less rendering power than virtual reality system [9]. Generally, visual AR systems use text, virtual 2D/3D objects or other visual information. For creating virtual 2D/3D graphical content, a rendering engine is required. The type of rendering engine depends on the system requirement. In an augmented reality system, if rendering performance is more important than the quality of the content, using rasterization renderer is more preferable. Most of the real-time augmented reality systems utilize this rendering technique as they need to deliver virtual content in a reasonable frame rate. Where high quality photorealistic contents are needed, usually rendering techniques based on ray tracing are employed. These techniques require powerful hardware which usually is not available on mobile devices. After generating the content, the output image is produced by compositing the augmented content and the input image of the real-world (Figure 5). Section 3 explains these two rendering techniques in more details.

2.2.3 Output

The output of the augmented reality systems can be visual or non-visual feedback. Sound, smell and haptic feedback are the examples of non-visual outputs [2], [10] that require special hardware in order to be presented to user. Nowadays, most AR focus on visual feedback. These systems use different types of display to present visual augmented content. In this section, common displays that are utilized in augmented reality system are discussed.

Generally, we can categorize augmented reality systems' displays into three groups which are *video-see-through*, *optical-see-through* and *video-projector*.

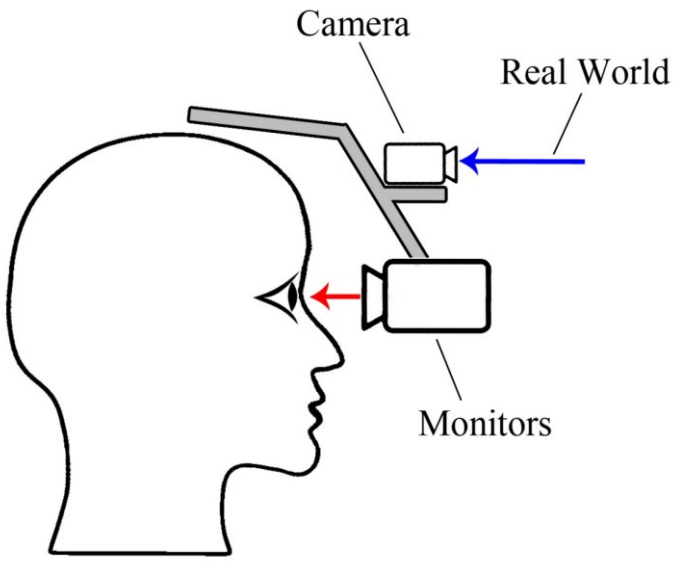


Figure 6. *Video-see-through.*

- ***Video-see-through:***
  The video-see-through is the most common display type in the AR systems. This type of display that is used in handheld devices, *closed-view head-mounted displays (HMD)* or monitor-based AR systems, provides an indirect view of the real-word [9]. The image of the physical word is captured by the device embedded camera or external camera. In the AR systems using this type of display, the real-world view and the generated augmented view are combined before presenting it to the user (Figure 6).

- ***Optical-see-through:***
  Optical-see-through displays are utilized by *Optical Head-Mounted Displays (OHMD)* and *Head-Up Displays (HUD)*. They are half-mirror displays which provide a direct view of the surrounding physical world. This feature is especially useful in some AR applications that the user may need to have a view of the physical-world even if the device is off [9]. In the systems using this display, the augmented content is projected on the half-mirror display in order to present both augmented view and real-world view to the user (Figure 7).

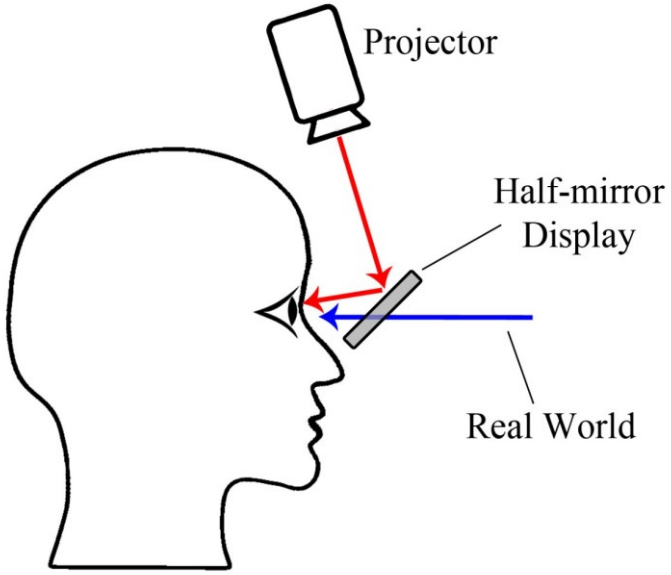


Figure 7.*Optical-see-through.*

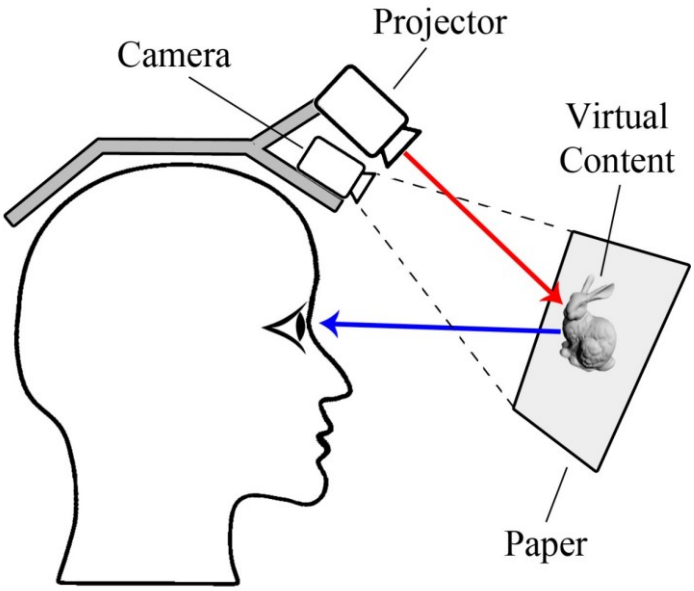


Figure 8. *AR Projector.*

- ***Video-projector:***
  Some AR systems use video-projectors so as to project digital information directly on the physical objects. In some of these systems, the display is separated from the users and they don't need to carry it with themselves. Instead, using different projectors placed in the environment, the augmented contents are presented to the users. Some of wearable AR systems are equipped camera and projector (Figure 8).

2.3 Pose Tracking Methods

In this section different pose tracking methods that are employed by augmented reality systems have been explained.

2.3.1 Vision-Based Tracking

In *vision-based* tracking, image data that comes directly from camera, recorded video or captured image is analyzed using different computer vision algorithms for pose estimation. This type of tracking has been more popular than other ones because for creating a simple AR system, just a regular camera is required [11]. Usually for sophisticated augmented reality systems, multiple cameras with different features, such as depth cameras, stereo cameras and fish eye lenses, can be used to extract more accurate data from the scene. The vision-based pose estimation methods can be classified into two groups, *Marker-Based* and *Marker-less*. These two methods are explained in more details in Sections 2.3.1.1 and 2.3.1.2.

Vision-based methods rely on a camera that captures image data. Generally, cameras' lenses have slightly different optical and physical characteristic such as focal depth, field of view, aspect ratio, optical center etc. These parameters are used to extract the information that are required for pose estimation, from 2D Images. The term *Calibration* refers to the determination of these parameters [12]. Precise calibration is important for an accurate pose estimation. Usually, calibration in an AR system is needed to be done once. In some cases that accuracy in the system is not crucial, we can use generic parameters.

2.3.1.1 Marker-Based Tracking

In the marker-based tracking, the position and orientation of the camera are calculated relative to the position of a unique marker in the physical environment. The physical environment should be marked with fiducial markers. These markers have unique patterns which are distinguishable from other objects in the scene. Fiducial markers are easily can be detected by computer vision algorithms and used to calculate position and orientation of the real camera.

Fiducial markers were first used by Rekimoto [13] in his augmented reality application. Since then many AR libraries and methods has been developed using these kinds of markers. Rectangular fiducial markers are the most common type of markers that are used for tracking in AR [14]. Pose tracking libraries such as ARToolkit and ArUco use these types of markers. There are different types of fiducial markers with different patterns used by detection technique for recognition [15] (Figure 9).

(a) ARToolkit *(b) ARTag* *(c) Intersense* *(d) Line Pencil* *(e) Pi-Tag*

Figure 9. *Examples of fiducial markers with varying patterns and detection techniques [15].*

2.3.1.2 Marker-Less Tracking

Although marker-based methods make pose estimation simple and fast, they are not suitable for all situations. It is not always practical to mark an environment with these markers beforehand, like unprepared outdoor environments. This limits the use of AR applications in many environments of interest. In these situations, other existing features in the scene should be used for Pose estimation.

Marker-less tracking refers to the tracking methods that use other distinctive information in scene instead of markers. In recent years many academic studies have been conducted on marker-less techniques which result huge increase in flexibility to adapt to different environments. On the other hand, immense advancement in computer hardware and computer vision allows the use of more sophisticated algorithms for tracking. In marker-less methods, tracking and pose estimation can be performed based edges, corners, model structure, texture of the objects in the scene or combination of them.

Currently, most of the practical marker-less AR applications use edge detection and matching. For example, Barandiaran and Borro [16] presented a real-time algorithm for 3D tracking based on edges detection and using a single camera. Lee and Höllerer [17] presented a method to detect human hands instead of marker and use fingertips tracking for pose estimation. Besides, there is a lot of recent work on marker-less tracking with RGB-D data, which simultaneously builds a 3D model of the environment. Newcombe et al. [18] in 2011, present *KinectFusion* which is a system for mapping real world scene accurately in real-time using Kinect sensor. They use all available depth data from the Kinect for tracking. KinectFusion type systems are simultaneously building a 3D model of the real environment, while tracking the camera's position within that environment.

2.3.2 Sensor-Based Tracking

Sensor-based tracking, as its name implies, calculates the camera 6DOF pose using sensors such as accelerometers, gyroscopes, digital compass, magnetic sensor, GPS, *Radio Frequency Identification Systems* (RFID). Pose estimation in these methods is performed

based on location and movement of the AR devices. Usually more than one sensors are needed to be used for more accurate tracking.

Tracking based on data from multiple sensors is called sensor fusion [19]. These days modern smart phones are equipped with some of these technologies that make using sensor-based tracking in AR systems more popular than before. There are plenty of successful services and games in the market using this type of AR system like “Pokemon Go” game.

2.3.3 Hybrid Tracking

Hybrid tracking method refers to tracking methods which use a combination of vision-based and sensor-based methods for pose estimation. This type of tracking is especially useful for some situations or environment that vision-based or sensor-based method alone is not practical.

2.4 Augmented Reality Devices

Augmented reality systems use variety of hardware devices based on their objectives. Some AR systems utilized hardware devices which have been designed specifically for AR, such as Google Glass and Microsoft HoloLens. Other augmented reality systems use devices which can provide components that an AR system needs such as smartphones and tablets.

In following sections, mobile devices such as smartphones and some of their components’ properties are explained, followed by brief introduction of Microsoft HoloLens in Section 2.4.2.

2.4.1 Mobile Devices

In this thesis, “Mobile Devices” refer to any hand-held devices which has camera and display, such as smart-phone, tablet, PDA and UMPC, excluding laptop.

Nowadays mobile devices are part of our daily life. We use them for various purposes from communication with each other to entertainment and education. These days, mobile devices utilize more powerful hardware which makes it possible to use them for more computationally demanding tasks. Augmented reality is one of the computer science fields which benefits from these hardware improvements. But compared to desktop PCs, they still have much limited resources that must be considered in mobile software development. In following part some important hardware components of the mobile devices are discussed.

- ***Processor*:**
  Parallel processing is main factor to achieve reasonable performance in real-time. Previously, mobile devices CPUs were designed with a single ALU and there was no multi-core or even superscalar technology. Thus they did not have parallel execution units [14]. Recently, mobile devices, like smart-phones, have architectures which can exploit all types of parallelism, including thread-level parallelism, instruction-level parallelism and data-level parallelism. Nevertheless, some limitations like memory bandwidth that affects the power consumption, are the bottleneck in used algorithm for mobile devices [20].

- ***Camera*:**
  Camera has become common feature in mobile devices. In most devices, especially in Android devices, the image from the camera is previewed in YUV formats which is also known as NV21. In order to save the image as JPEG, for example, or using in some applications it must be converted to RGB.

- ***Display*:**
  The other characteristics of mobile devices is their display size and ratio. Due to different size of devices, the displays may have different size and ratio, which must be considered in rendering the output image. 4:3, 16:9, 3:2, 16:10 and 17:10 are the common ratio that mobile displays are made based on. Displays with same ratio may have different resolutions.

- ***Sensors*:**
  Most of the modern mobile devices are equipped with several sensor devices such as IMU and GPS. These sensors can be utilized in augmented reality application for more accurate and robust tracking and pose estimation.

2.4.2 Microsoft HoloLens

Microsoft HoloLens is one of the most recent wearable mixed-reality smart-glasses devices that is available in the market. It is a standalone computer with Windows 10 operating system and includes multiple sensors, different kind of cameras, holographic lenses and speakers. Therefore, it able to perform all tasks of an AR system independently, without any cords or extra attached devices.

HoloLens understand its surrounded real-world environment from obtained data by an array of cameras and sensors contained in the device. These sensors and cameras are: Ambience Light Sensor, Depth Camera, HD Video Camera, and Environment Understanding Cameras.

The HoloLens is an “optical see-through” system with “hybrid tracking”. In order to display the augmented reality, HoloLens uses holographic lenses which are placed in front of the users’ eyes. These lenses are high-definition stereoscopic 3D optical head-mounted display (OHMD). The virtual content is projected on the lenses and, user is able to see the real world through them as well.

HoloLens allows user to interact with virtual environment using Gaze, Gesture and Voice input that make interaction more natural. Gaze indicates user’s attention point in the scene. The direction of the gaze is indicated with a cursor which follows the user’s point of attention when user’s head moves. User can interact with the virtual objects using gesture based on gaze direction. Voice command is the other user input that provide user interaction with virtual world. The user’s voice command also applies to the virtual object that is pointed with Gaze. Voice command could be a command to the system or to do a customized action.

3. OBJECT RENDERING IN AUGMENTED REALITY

This Chapter focuses on rendering 3D virtual object in augmented reality and some aspects of photorealistic approaches. Therefore, we start with definition of Rendering in computer graphics and two famous rendering methods which are *Ray Tracing* and *Rasterization*, followed by shadow techniques. Then some practical techniques for implementing caustics effect using rasterizer is discussed. Finally, light direction estimation techniques and approaches, in augmented reality are explained.

3.1 Rendering

The process of creating a 2D image of a 3D scene or model is called rendering [11, p. 1]. In rendering process, the color of the 2D image's pixels is specified by computing the light intensity of the scene corresponding to each pixel.

In general, the result of the rendering process can be categorized into two types which are *photorealistic* and *non-photorealistic*. Photorealistic rendering tries to generate an image of a 3D scene which is identical to a photograph from that scene in the real world as much as possible. In order to do so, all type of the light effects in the real world, such as reflection, refraction, shadow, caustics and indirect illumination, are tried to be simulated. Non-photorealistic rendering, by contrast, produces an image of 3D scene which looks like a drawing from same scene.

Many different rendering methods have been developed which can be categorized in two groups, ray tracing and rasterization. Each of these methods have their own advantages and pitfalls that make them suitable for different purposes.

3.2 Ray Tracing

Arthur Appel [21] introduced Ray casting, which is a point-sampling rendering technique, in 1968. In the real world the light source emits photons to the scene which are reflected, refracted or observed. In the ray casting technique, in contrast to the real world, a ray is cast from the center of a virtual camera or eye through the scene for each pixel of the image. The path of this ray is traced through the scene until it hits an object. If the primary ray does not hit any obstacle, the color of the pixel is set to the background color. If the ray hits more than one object, nearest one to the camera is considered as the collided object. After finding the intersection point of the ray and the collided object, the color of the pixel corresponding to the intersection point, can be computed using properties of the object material and the light source. For the diffused surface, the light intensity in the

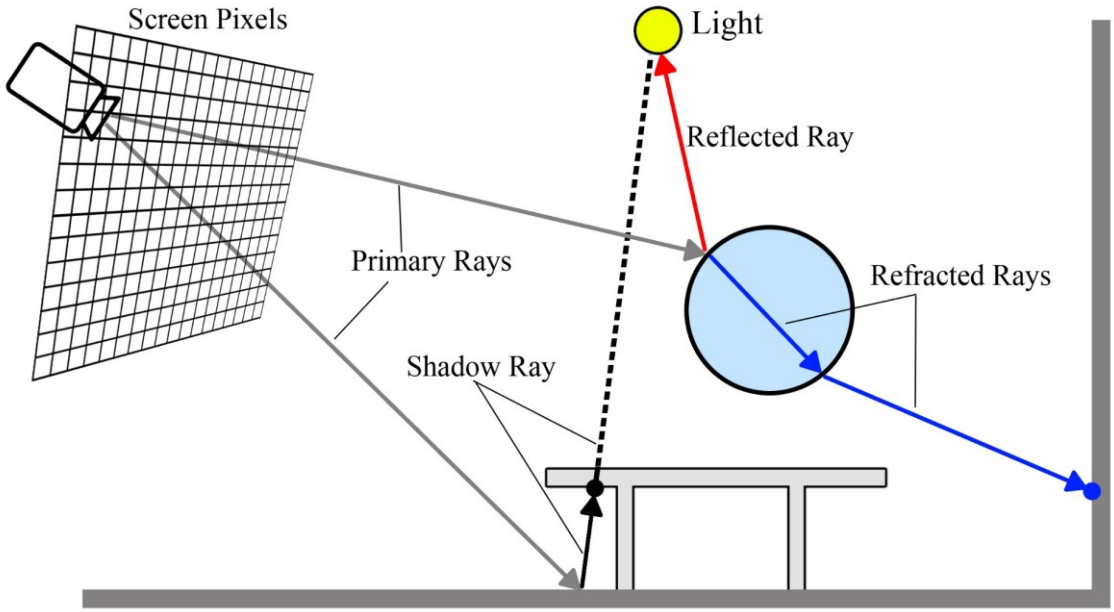


Figure 10. *Ray tracing.*

intersection point can be calculated based on Lambert's cosine law. In Lambert's cosine law the light intensity in the intersection point is directly proportional to the cosine of the angle between the light source direction and the surface normal.

$$Ir = Ip * cos\,\theta$$

Where:

Ir = Light intensity in the intersection point

Ip = Intensity of the point light source

θ = Angle between surface normal and light source direction

The ray casting technique is easy and straightforward to implement but it does not compute shadows, reflection and refraction which are important effects for creating photo realistic image.

In 1980 the ray tracing algorithm was introduce by Whitted [22]. Ray tracing is an extension to the classic ray casting and tries to overcome the ray casting weaknesses [23, p. 4,37]. In contrast to ray casting [21], for computing shadow, reflection and refraction, the collision detecting between a ray and objects does not stop when closest hit object is found (Figure 10). In this algorithm when a primary ray hits a surface, a ray is cast in the direction of light used to determine if there is some object between the source and the primary ray hit-point, casting a shadow. When the primary ray hits a specular or mirror

surface, a secondary ray is generated and traced from previous ray intersection point to the next surface. The direction of this new ray is calculated using reflection equation. If the primary ray hits a transparent object, like glass sphere, a new ray called refraction ray is generated. This refraction ray is traced through the object. This process is continued recursively until it hit a diffused surface or reaches the maximum number of iteration that ray can be traced. Then color of the image pixel can be computed by averaging of these rays' contribution (reflection/refraction/shadow) (Program 1).

```
Render image using ray tracing:
    for each pixel
        pick a ray from the eye through this pixel
        pixel color = trace(ray)

trace(ray)
    find nearest intersection with scene
    compute intersection point and normal
    color = shade(point, normal)
    return color

shade(point, normal)
    color = 0
    for each light source
        trace shadow ray intersection to light source
        if shadow ray intersects light source
            color = color + direct illumination
        if reflective or transparent
            color = color + trace(reflected / refracted ray)
    return color
```

Program 1. *Ray tracing pseudocode [23, p. 37].*

The Whitted's ray tracing algorithm cannot simulate all kind of light scattering (caustics, motion-blur and indirect illumination). Other rendering algorithms based on original ray tracing have been introduced such as *Path Tracing* [24], *Bidirectional Path Tracing*, *Distribution Ray Tracing* [25] and *Photon Mapping* [26] to simulate these effects which result more accurate and realistic rendered image [23, p. 4].

Kajiya [24] introduced a mathematical formula that describes global illumination in 1986, which is known as the Rendering Equation. All global illumination algorithms try to simulate all kinds of light scattering by solving this equation.

$$I(x,x') = g(x,x')[\epsilon(x,x') + \int_S p(x,x',x'')I(x',x'')dx'']$$

In which:

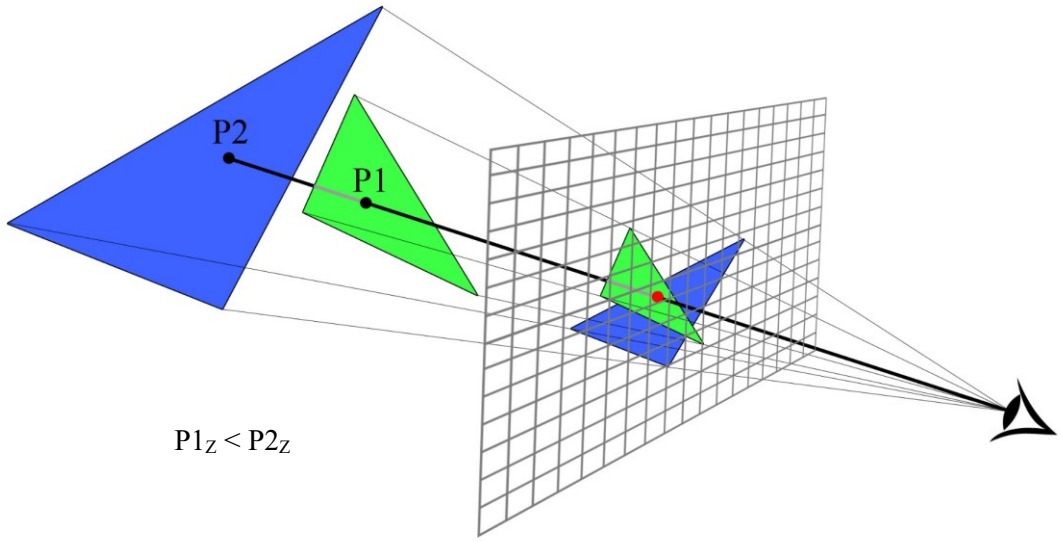


Figure 11. *Rasterization - surfaces with less Z value are visible.*

- $I(x, x')$ is related to the intensity of light passing from $x'$to point x
- $g(x, x')$ is a geometry term
- $\epsilon(x, x')$ is related to the intensity of emitted light from $x'$to point x
- $p(x, x', x'')$ is related to the intensity of light scattered from x'' to x by a patch of surface at x'

3.3 Rasterization

Rasterization, like the ray tracing technique, is used to create a 2D image from a defined 3D scene but in different way. In ray tracing, visibility of objects' surface in the scene are checked for each pixel of the image plane (screen space). In rasterization, all image plane pixels that are covered by objects' surface, where the objects in the scene are usually constructed by triangles, are determined and will be colored based on the object's color. In order to do so, each objects' triangle in the 3D scene is projected onto screen space then the pixels of the image plane which are in the projected triangle, will be determined. In the rasterization method, visibility of two intersected triangles is often calculated by Z-Buffer. The triangle with less value in Z-Buffer would be visible on the screen (Figure 11). It can be also, be done by pre-sorting the triangles, especially it needs to be done this way if there are transparent objects.

In rasterization, the triangles are independent of each other and they can be processed independently. This feature makes this method, like ray tracing, massively parallelizable. GPUs which are in todays' desktop computers and mobile devices use rasterization to render virtual 2D and 3D scenes. They are heavily optimized for this method and leverage its parallelizability feature to speed rendering process. Modern GPUs can process hundreds million triangles per seconds which allows to create complex scene in real-time. In general, when performance is more important than accuracy of the rendered image, the rasterization method is preferable than techniques which are based ray tracing. These days, this method is widely used for real-time rendering and interactive application like video games and virtual reality software.

Although rasterization provides good performance for real-time rendering, it cannot produce some light effects like refraction, reflection, shadow, caustics and indirect illumination properly that are crucial for physically-based rendering. Usually different tricks and techniques are used to simulate these effects approximately. For example, using shadow mapping and reflection mapping to create these effects in the virtual scene. Also for creating some of the light scattering, such as caustics and indirect illumination, it is necessary to use other methods along with rasterization. This makes rendering process more complicated and inefficient, and usually the final result is not accurate and realistic. Albeit ray tracing based methods are time-consuming, they can produce high quality image by simulating all light scattering. Because of that these techniques are more suitable for physically-based rendering rather that rasterization.

Several graphics APIs such as OpenGL, Direct3D, Vulkan and Metal have been developed that use the rasterization methods for 2D/3D rendering. Some of these APIs only support specific platform/operating system, such as Direct3D (Microsoft Windows) and Metal (iOS, macOS), and other such as OpenGL and Vulkan are cross-platform. In this thesis work, OpenGL (and OpenGL ES) has been utilized for rendering.

3.4 Shadow

The shadow effect provides right perception of a 3D virtual environment for the viewers. Correct shadow of the virtual objects will define the relation of the objects in the scene with the other objects in terms of distance and size [27] (Figure 12).

Figure 12. *Shadow presents the position of the bunny from the ground. The image on the left side without shadow doesn't show the position of the object (adapted from [27]).*

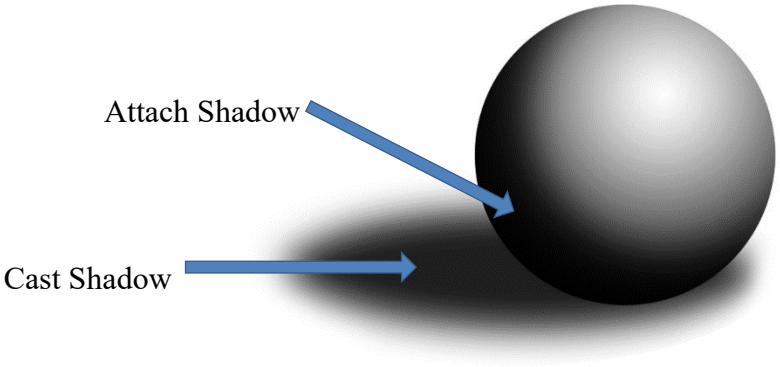


Figure 13. *Cast Shadow vs Attach Shadow.*

There are two types of shadows, which are cast shadows and attach shadows [28] (Figure 13). Attach shadows appear on parts of an object where the surface normal is pointing away from a light source. Whereas, cast shadows will occurred when an opaque object is placed between a light source and a surface, then the surface is shadowed, where the normal of the surface is facing the light source. Also, cast shadows on an object can happen when its shadow falls on the object itself.

Since in AR, normals of the virtual objects and virtual light source position are defined in virtual world and are not related with the real word, attach shadows can be produced easier. On the other hand, cast shadows related to all real and virtual objects in the AR rendered scene. Cast shadows can be grouped in four categories based on type of that

shadow caster and shadow receiver. Shadow may cast from real object on real object, from virtual on virtual object, from virtual object on real object or from real object on virtual object.

The first two groups that both shadow caster and shadow receiver are in same world, either real or virtual world. In case of real objects in real world, cast shadows occur naturally. In virtual world, cast shadows from virtual object on virtual objects can be computed based on shadow algorithms in CG.

Main challenging part of shadows in AR is the other two cast shadows groups, where shadow caster and shadow receiver are from different worlds, one is in real world and the other one is in virtual world. One of the proposed solutions to achieve these two groups of cast shadows is generating a virtual model of the real object as a phantom object [29]. All cast shadows can be obtained using the phantom and virtual objects in virtual world and will be added to the image of the real world.

There are two ways to reconstruct the real-world scene in virtual-world so as to us it as a phantom environment. One way is re-modeling the real scene with simple virtual objects [30]–[32]. The main challenge using this technique is placing the phantom object in correct position of the related real object in the scene, especially in dynamic scene. Pessoa et al. [31] and Santos et al. [32] use separate markers to track a real object that the phantom object is assigned to. But it may not be practical to apply for every existing object in the captured scene with numerous objects.

The other way to reconstruct the real world is using RGB-D data of the scene. There are different technologies for RGB-D imaging, such as Depth cameras, RGB-D cameras, *Time-of-Flight* (ToF) camera, structured light, flash lidar and Stereo Cameras. By using these cameras, the scene can be scanned and a virtual model of the real world can be reconstruct based on the RGB-D data [18]. Eigen et al. [33] utilized machine learning method to achieve depth mapping the captured scene in real-time from a single image. They used two deep network stacks. The first one makes a large global prediction of depth data from the whole image and the second one refines the global predicted depth data locally.

In rasterization, shadow mapping and shadow volume are the most used shadow algorithms in real-time rendering. The shadow volume is introduced by Crow in 1977 for the first time [34]. In this technique, first the shadow volumes (stencil shadows) which are a 3D region of the scene that is occluded from the given light source, is calculated (Figure 14). Then shadow will be rendered for any geometry of the scene that intersects with the shadow volumes.

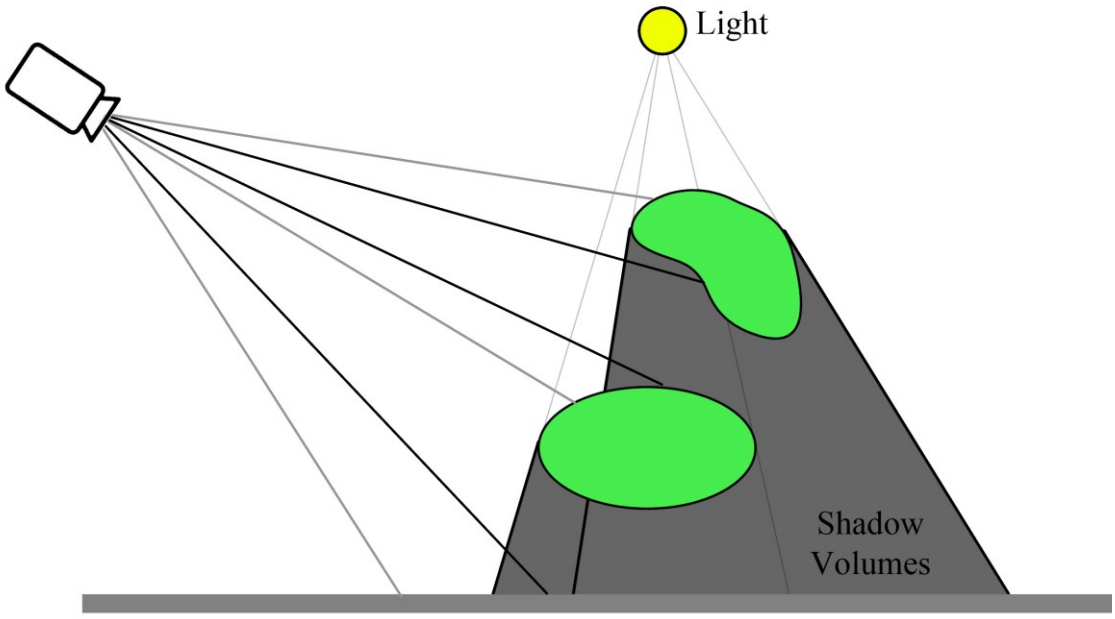


Figure 14. *Principle of the shadow volume algorithm.*

Compared to shadow mapping, shadow volume technique generates much less noisy shadows, but since it must be done for every shadow casting object for every light source, it is much slower.

The shadow mapping [35] is a two-phase technique. In the first phase, the *shadow map* is created. The shadow map is a buffer that keeps the depth value (Z value) of each pixel. In practice, a texture object is used to store the depth values in the GPU texture memory. Each value stored in the shadow map indicates distance between the object in the scene and the light source, for a specific pixel. In order to create the shadow map, the scene is rendered from the light source point of view and then instead of the color value, depth value of each pixel is stored in the shadow map. In the second phase, the scene is rendered from the camera point of view. In this phase, the shadow map is used to see whether a pixel is in the shadow or not. For this purpose, first the coordinate of the rendered pixel is converted to the light source coordinates, then if the depth value of the pixel was greater that the stored value in the shadow map, the pixel is in shadow (Figure 15).

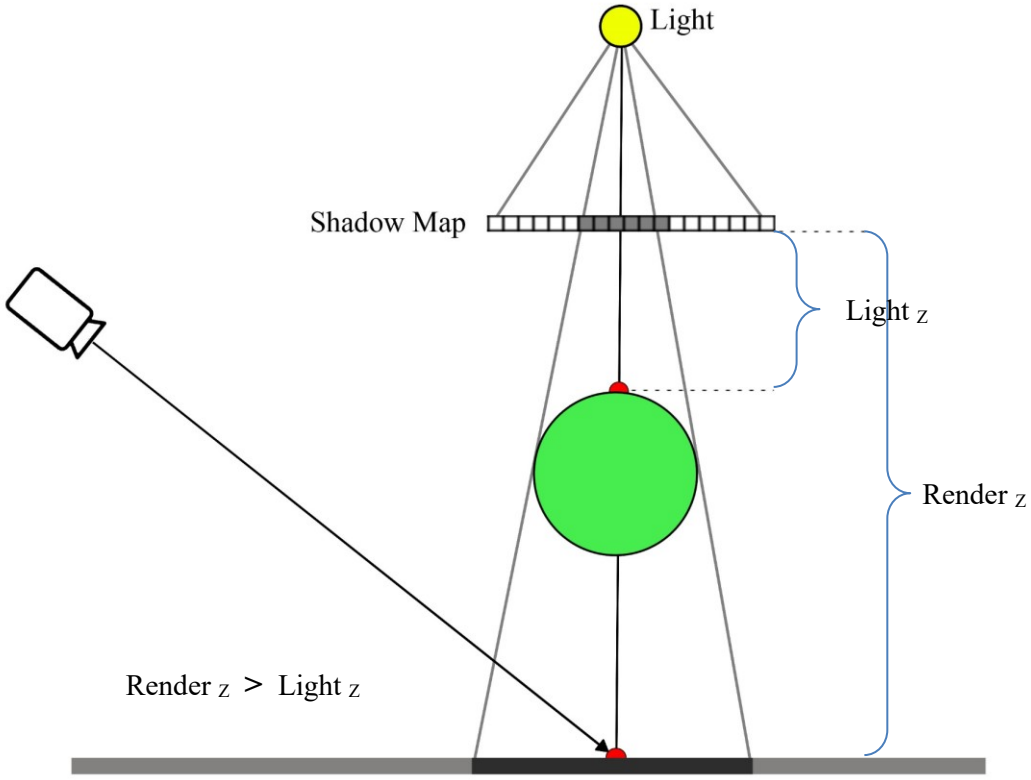


Figure 15. *Shadow mapping illustrated.*

3.5 Transparency and Reflection

Some illumination effects such as refraction and reflection, on shiny and/or transparent objects are main factor of realism in rendering. These phenomena can be achieved in different way based on chosen rendering techniques, ray tracing or rasterization.

Ray tracing is able to provide most of the main illumination effects with more accuracy than rasterization. By using ray tracing, reflection and refraction can be computed for many ray bounces. One of the specular effects is caustics that can be rendered using photon mapping algorithm, which requires tracing photons from light source/s, pass though transparent objects and bounce on all type of object for defined number of paths, therefore it is based on ray tracing. On the hand, rasterization is much faster, but because of its limitation it is inaccurate in these light illumination effects.

In order to perform reflection effects, it needs to have 3D information about the surrounding area. In case of virtual objects in the scene, the information exists. However, captured image from the real world is just a 2D image of the scene which would be rendered as the background. Therefore, when a ray is reflected, it may not hit any surfaces and nothing would be reflected on the shiny objects.

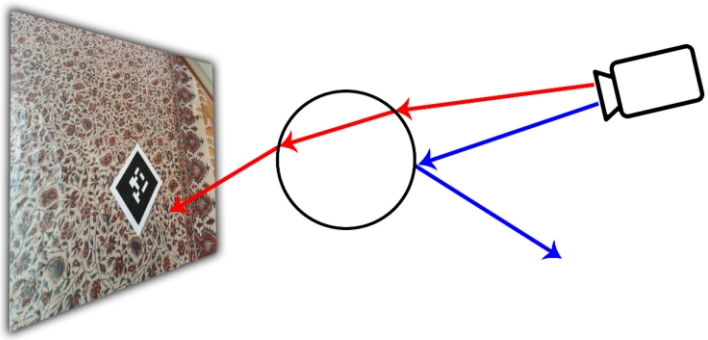


Figure 16. *Reflection (blue arrows) and refraction (red arrows) in AR with flat image input.*

Refraction effects can be almost simulated based on captured image of the real world. Since refracted rays go through the object and the scene behind the object would be visible, in AR we can use the captured image as the scene behind the transparent object and present refraction effect roughly. Reflection and refraction matters in AR is illustrated in Figure 16.

In order to render convincing reflections, an AR system needs omnidirectional information about the environment, e.g., an environment map captured with a light probe. In the absence of a 360-degree environment map, it is possible to obtain reasonable visual quality by stretching a input image with a narrow field of view into an environment map [36].

As it is mentioned earlier, in order to render caustics, using the ray tracing algorithms, such as photon mapping is required which is time consuming and not suitable for real-time rending. On the other hand, due to limitations of the rasterization, achieving caustics using rasterization is much complicated and inaccurate. Nevertheless, Shah et. al. [37], used combination of the shadow mapping technique and photon mapping, which they called "*Caustics Mapping*", to implement approximate real-time caustics rendering using rasterization.

3.6 Photorealistic Rendering

Compared to fully virtual content, there are special challenges for photorealistic rendering in AR. Essentially, we have imperfect information about lighting, scene geometry, surface materials and the camera model in the captured scene in the real world. Therefore, these must be either estimated, which could be very difficult, or use some kind of approximations. In this section, some of the related works of photo-realistic rendering in augmented reality, are outlined and discussed.

In 1992, Fournier et al. [30] proposed a method for first time to compute global illumination. They reconstruct the scene using boxes and substitute them for the real object in the scene for computing the global illumination. Then they render the virtual objects in the video sequence with real world background images. In order to compute the global illumination, *progressive radiosity* method is used.

Agusanto et al. [38] (2003) use image based lighting and environment illumination maps to calculate global illumination for photo-realistic rendering in AR. They define different environment illumination which are glossy and diffuse environment map. In order to compute the specular and defuse components from an environment map that is achieved from light probes in the scene, they use environment map pre-filtering and radiance. For rendering in AR, they use multi-pass rendering algorithm. With this technique they could achieve almost real-time rendering with approximately 17 FPS.

Pessoa et al. [31] in 2010, also, used image based lighting where for each synthetic object in the scene, they generate different environment maps which have different glossiness levels. Besides that, they used spherical harmonics transformation to generate a uniform environment map that can be applied for every virtual object. In order to achieve more photo-realistic result they combined *Lafortune Spatial BRDF* (SBRDF), Fresnel effect and tangent rotation parameterization, which is their invention in their technique. Their AR application supports a number of complex visual effects, including: Color bleeding, occlusion, Fresnel term, normal mapping, refraction, retro-reflective material and interactive reflection.

In 2012, Santos et al. [32], presented a "Real Time Ray Tracing for Augmented Reality" pipeline named RT^{2}AR. Since processing each pixel in RT^{2}AR is done independently, they are able to integrate their RT^{2}AR pipeline with image-based tracking techniques easily. They used ARToolKitPlus library as pose tracker and the Microsoft Kinect to get depth image data of the scene. By RT^{2}AR they could achieve real-time rendering with performing visualization and illumination effects, such as soft shadows, reflection, occlusion, custom shaders and self-reflection that occur with interaction between real and virtual objects in the scene. In order to obtain more photo-realistic effects that related to interaction of real and virtual objects, they remodeled some real objects in advance. Also, using depth information from Kinect allows to avoid remodeling real object in some cases.

In the same year, Kán and Kaufmann [39] also used ray tracing in rendering for augmented reality to achieve high quality results in light illumination effects rendering such as reflection, refraction and caustics. Photon mapping method is used for simulating caustics effect. For calculating caustics that can be created by refracted or reflected light on virtual or real objects, they proposed a method that can be performed in interactive frame rates. Compared to previous works, the advantage of Kán and Kaufmann's work is ability of rendering the specular surfaces (e.g. mirrors and glasses) very naturally. This achievement allows them to present a novel photo-realistic rendering for AR. They used fish-eye

camera to obtain hemispherical environment-map. The environment map texture is used whenever the traced ray doesn't hit any virtual or real geometry. In addition, they studied their system's ability based on users' perception of the rendering results.

Croubois et al. [40], implemented a realistic AR on mobile device (iPad Air) using a simple version of image based lighting method in 2014. they presented an environment map acquisition method where the environment map is updated dynamically using cell-phone front camera. Then they use the environment map to light the synthetic object in the scene. Their solution is able to handle soft shadows and dynamic environments, assuming to use Blinn-Phong BRDF, distant lighting and planar local geometry.

One of the most recent work related to photo-realistic rendering in augmented reality is Rohmer's et al. [41] work that was published in 2017. They presented a pipeline with two stages for environment acquisition and augmentation on mobile device with a depth sensor. In this pipeline, first they capture the environment using input data such as *Low Dynamic Range* (LDR) images, estimated camera 3D poses and depth data from the mobile device. The captured environment will be stored as a point cloud which be used to render G-Buffer [42]. Then they use push-pull steps [43] to fill holes of the captured environment map. Also, this buffer contains estimated normal and the screen space depth. Besides, they presented a linear estimation of the unknown color adjustments that the driver and mobile camera apply to the output image. By this estimation, they can merge LDR color information of consecutive frames with high dynamics range samples.

The obtained information and a RGB-Depth environment map which is called "*Distance Impostor* (DI)", are used as the input with several techniques that they present, for rendering virtual objects with natural illumination effects. They use combination of GPU Importance Sampling [44] and Impostor Tracing [45] to perform an image-based ray tracing variant, for rendering part. They implement their solution with three different implementations, which are *Voxel Cone Tracing* (VCT), standard *Environment Mapping* (EM) and *Distance Impostor Tracing* (DIT) and compare them in terms of quality and performance. The final result is an augmented reality with occlusion, perspective, shading and the original color transformation of camera with white balance and exposure.

3.7 Caustics Implementation

Caustics is one of the light scattering effects that has a key role in photo-realistic rendering. Due to the characteristics of this light effect, best results can be achieved using the ray tracing techniques such as photon mapping. Caustics occurs when light photons are focused due to refraction or reflection by a curved specular object or surface. This effect may occur after one, two, or more iterations of refractions or reflections. Each iteration requires a set of calculations so as to trace the light ray and check whether caustics occurred or not. Some techniques such as photon mapping facilitate this approach. On the other hand, implementation of ray tracing for multiple iterations using a rasterizer is not

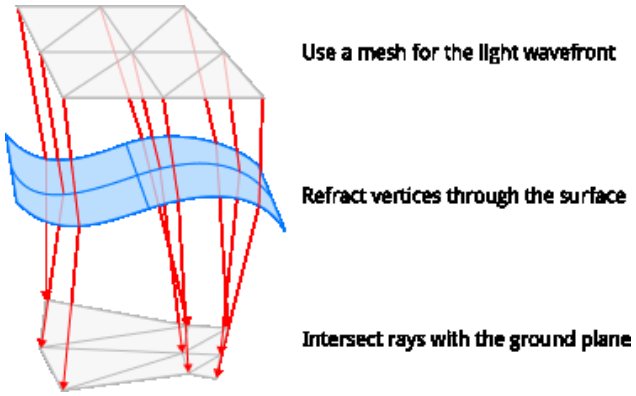


Figure 17. *Using mesh for approximating a wavefront of the light [46].*

efficient. Rather than using the ray tracing techniques, there are some techniques that can be used to simulate caustics using rasterization in real time. These techniques are dependent on the number of iterations of refractions or reflections.

Rendering caustics under water is an example of the effect that occurs after the first refraction iteration. Wallace [46], used meshes to approximate a wavefront of the light. In this technique, each ray of a direct light is represented with a vertex of the mesh and the area of a triangle represents equal number of light rays within the area of a triangle. Therefore, a triangle with a wider area denote that light is spread out which results in less brightness. In contrast, a triangle with smaller area means more brightness. After refraction of the vertices through the water surface and intersection with a ground surface, their positions are changed which causes variation in the areas of the triangles (Figure 17).

Since this technique uses area of triangle to represent the light density instead of sampling, there is no need to use a huge number of samples to calculate the caustics and it makes it more efficient. However, this technique is suitable for special cases where the caustics occurred after just one refraction iteration through a planar body of water. In order to render caustics that happened due to refraction of light through a glassy object, it requires to trace the light ray with at least two iterations of refractions. McGuire et al. [47], introduced *Image Space Photon Mapping* (ISPM), which uses image space techniques to accelerate photon mapping. In this technique, first a screen-space deferred-shading G-buffer [42] is created by rasterizing the scene from the eye. Then the bounce map is created for each point light source. The bounce map is similar to a shadow map, which maps the first bounce hit point of emitted light ray in light space. Next step is the second pass of tracing the photons from the bounce map and then creating a photon map. Finally, from the eye, the scene with the indirect illumination is rendered by rasterizing photon volumes. Reznik used ISPM to simulate caustics using a rasterizer and walkthrough the implementation caustics and refraction practically [48].

3.8 Light Direction Estimation

Accurate estimation of light direction results precise light illumination to achieve more realistic augmented reality. Several methods have been proposed in order to estimate light direction in captured real scene [49]. In this section some of these methods are explained.

Light can be estimated directly from environment map. In order to have good lighting, *High Dynamic Range (HDR)* environment map is needed. The taken HDR image with just one exposure time, the bright areas in the photo get clamped to maximum brightness, which causes losing information. Multiple images have to be taken of the lighting probe with different exposure times so as to have accurate HDR environment map [50].

Debevec [51] presented direct estimation of light method by using light probe to measure the incident light at the location of the synthetic objects. This method uses high dynamic range image based model of the scene to compute an environment map and light direction directly based on light probes placed in the environment. The image of the environment can be obtained using a full dynamic range photograph of a reflective object (light probe), such as mirrored ball, that is placed in the scene and captured by camera [51].

Light probes are able to reflect an image of the environment at full frame rate and environment map can be obtained using reflected image on a light probe. This significantly required less work than scanning the surrounding environment using hand-held camera. However, using light probes are not suitable for casual use. In case of static lighting, environment map can be achieved once with an off-line scanning pass [52]. One way to capture environment image in all direction is using omni-directional camera or cameras with fish-eye lens [53], [54]. By using cameras with these kind of lenses, environment image can be obtained in on step and it helps to contract the environment map easier and faster.

Light estimation from specular reflections is the other method of light direction estimation that can be used without having light probe or environment map. In this technique, light direction is estimated from detected reflected light on a known specular object. This concept can be applied on any known specular shape in the scene, and it is not limited to just light probes. Based on this principle Lagger and Fua [55] recover light sources and their directions in a scene by detecting specular highlights on moving objects. Mashita et al. [56] estimate point light direction by detecting specular highlights on planar objects. Jachnik et al. [57] use moving a hand-held camera around a small planar surface, such as a glossy book cover to capture a 4D surface light-field. They split the surface light-field into two the view-independent diffuse and the view-dependent specular components.

In a scene without any specular object, it is possible to estimate light direction from diffuse reflections. Since incoming lights from different direction to a diffused surface must be distinguished, estimation of incident light direction would be much difficult.

Gruber et al. [58] present an interactive light estimation based on spherical harmonics (SH) [59] and using RGB-D camera. First, the scene is reconstructed based on the input depth images. Sphere has the ideal surface for estimating the light direction. Therefore, some sample points are selected from the surfaces in the reconstructed scene, which have suitable surface normal vector distribution. Then light direction is estimated for the selected sample in spherical harmonics form. Reflections on diffuse surfaces collect incoming light from every direction. Therefore, for each sample point, they must compute the shadows of every other objects that are in the scene.

Boom et al. [60] also, only use RGB-D camera to estimate the point light source position in the scene, in different way. They use information from depth camera (Kinect) to obtain the normals of the objects and assume that their appearance can be roughly described by Lambertian reflectance model. Then the RGB image is segmented by color where each segment has similar color and represent approximately similar albedo. By using these data, they propose an interactive search method to estimate point light position.

Uranishi et al. [61] propose a light direction estimation method using structural color. They use special marker called *The Rainbow Marker* which is a planar marker and has microscopically structured surface. This kind of marker produces structural color and used as light probe in the scene. Depending on viewpoint, the spectrum and direction of the light source, the produced structural color would be different. The input color pattern is checked with predefined color patterns to estimating the light direction based on the matched pattern. They provide two prototypes of the rainbow markers. One of them is made with a holographic sheet and the other one is made with a grating sheet. In order to identifying these markers in the scene, they suggest using ID markers attached to the rainbow markers.

Light direction in a scene with light illuminations that affected by sunlight, specifically at outdoor scene with daylight, can be estimated based on the position of the sun depend on date, time and the position of the scene on the Earth.

Madsen et al. [62], estimate light direction at outdoor scene using provided data by the camera, such as date, time and the position on the Earth (from GPS information), that is stored in the output image header. Also, they use an automated shadow detection process and use obtained information to estimated sky and sun illumination. In their work, there are three main assumptions that are required to make their method work. First, available required information about date, time and GPS data for the input image. Second, the input image is an outdoor scene image with only natural illumination, that in this case it would be sun and sky illumination. And the last assumption is, most of the surfaces in the scene be almost diffuse.

Madsed and Lal [63], besides using GPS information, date and time to estimate sun light direction at the outdoor location, they use depth camera and detecting shadows of the object from the depth image to enhance illumination estimation directly from input image

of outdoor scene without any light probes in daylight situation. First, they detect moving objects, such as car, people or tree leaves, in the scene by tracking color and depth data from the stream input. Then based on the data that is the combination of depth information of the scene and color information of the detected shadows, they estimate illumination of the real scene, the color and intensity of the sky and the sun. Also, they use [64] to estimate the sunlight direction.

Castro et al. [64] also used same technique to estimate the sunlight direction using GPS data, date and time at the outdoor scene. They use iPad2 device which is equipped with camera and GPS that can provide required information for estimation of the sunlight direction.

4. RELATED WORK

There are dozens of commercial, free and open-source AR SDKs available. In this Chapter we have explored the ARToolKit, ARToolkitPlus SDK (which is chosen for this thesis work), ArUco and two new AR APIs, Google ARCore and Apple ARKit, that are released recently.

4.1 ARToolkit

ARToolkit is one the first vision-based tracking library which was developed by Hirokazu Kato [65] for creating augmented reality applications. This open-source library was released under GPL license in 1999. Since then different versions of the ARToolkit have been developed for different operating systems and platforms, including web and mobile devices. The earliest version of this library just used square fiducial markers for pose estimation. The position and orientation of the camera are calculated using predefined physical marker in the scene. The ARToolkit uses *template-matching* to detect the markers. In this technique user needs to define pattern of the marker for the system beforehand. The ARToolkit provides a utility for enabling the users to define a new marker pattern. During the run-time, the input image is compared against the defined marker pattern to detect the marker in the scene.

The ARToolkit provides a cross-platform API (libARvideo) for capturing image data from different sources. It supports different video formats such as YUV and RGB. Depending on the used platform and the user's preference, the libARvideo uses different video libraries for capturing image data. For example, for Windows OS, the Microsoft DirectShow and for the Linux OS, Video4Linux (V4L) or the GStreamer framework can be utilized.

The ARToolkit SDK comes with a set of utilities for calibrating the AR system camera that support both single and stereo cameras. It is also possible to use the tool that provided by the OpenCV to calibrate the system camera. The ARToolkit also offers a utility which allow to get calibration data of a specific camera from an ARToolkit camera calibration server during the run-time for camera calibration. It is especially useful when we do not want to force the users to perform the camera calibration by themselves.

The ARToolkit provides a set of functions and utilities which facilitate using OpenGL for 2D/3D rendering in desktop and mobile (Android and iOS) AR applications. In earlier version, the OpenVRML was utilized for loading and rendering 3D models. Later, support for the OpenSceneGraph library has been added to the ARToolkit as well. This library is used for advanced and efficient 3D rendering. Also, several ARToolkit plugins for different rendering and game engines have been developed, such as Unity 3D [66].

Due to good community and strong support, ARToolKit has been evolved and new features such as *Natural Feature Tracking* (NFT), pose filtering and support for stereo cameras have been added to this AR SDK in recent years.

4.2 ARToolkitPlus

ARToolKitPlus is pose tracking library which is presented as a successor to ARToolkit. It has been optimized and developed to be used on mobile devices with weaker processing power such as smart-phones [14]. ARToolKitPlus is an open-source project that has been released under GPL license in 2006. It is lightweight and fast and can be utilized on old mobile devices with reasonable performance. It has been used in AR application developments, nonetheless, it is not developed anymore since 2014.

In contrast to original ARToolKit that has been implemented using C language, developers of ARToolkitPlus used C++ to implement it in object-oriented paradigm, thus it can be maintained easier. Although this library is based on ARToolKit, its developers considered some fundamental modifications, to make it suitable for low-end mobile devices. For example, using fixed-point arithmetic instead of floating-point one because floating-point operation slow down computation on the mobile devices without *Floating-Point Unit* (FPU). The ARToolkitPlus supports both the template-markers and ID-markers [67]. These two types of markers have some similarities and differences. Both template-markers and ID-markers are black and white with distinctive black border which make it easier to detect and track them. Template-markers have simple image as the marker template in the middle. In contrast, ID markers include black and white squares in the middle area. These squares define a binary number. Usually simple ID markers include small data such as an ID number.

Using ID-markers is more favorable because the users do not need to define the marker to the system and can use ID-markers which are defined. Also, using ID-markers instead of template-markers yields better performance as image matching is not needed for detecting ID-markers. The ARToolkitPlus can use native pixel format provided by mobile devices camera so format conversion is not required for tracking [14].

The process of pose estimation in the ARToolkitPlus can be divided in several stages which are [14]:

1) ARToolkitPlus receives supported input image.
2) Input image is converted to the binary image using thresholding algorithm.
3) The existing markers in the scene are detected using template matching or ID-marker detection algorithm.
4) ARToolkitPlus undistorts detected markers.
5) The pose estimation is done for each detected marker.

Figure 18 shows the simplified data flow in ARToolKitPlus.

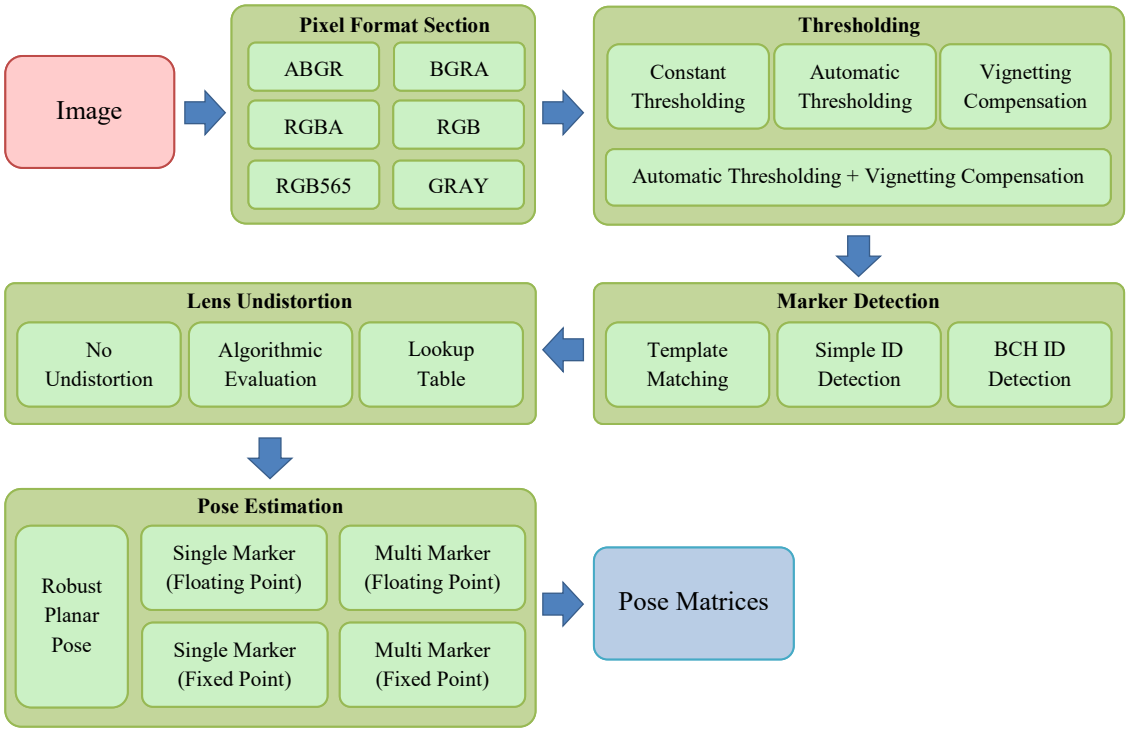


Figure 18. *Simplified data flow in ARToolKitPlus (adapted from [14]).*

In contrast to the original ARToolKit, ARToolkitPlus is just a pose tracking library and does not depend on any external library. This feature makes this library easily portable for multiple platforms. ARToolKitPlus can be utilized for different applications that need visual tracking such as augmented reality applications and robotics localization. This library does not provide any functionality for 2D/3D rendering and accessing camera devices, therefore the AR applications that utilize the ARToolkitPlus should handle them.

4.3 ArUco

ArUco is an open-source marker-based pose tracking library which has been developed at Cordoba University and released to the public under BSD license. This library has been implemented in C++ and rely on OpenCV library for image processing [68].

The ArUco uses the square-based fiducial markers for camera pose estimation. This library has its own set of markers (markers dictionary) named *ARUCO_MIP_36h12*. The markers of the ArUco consist of a black border and inner region with a unique pattern. This pattern specifies the identification of the marker. The ArUco also supports the fiducial markers that used by popular augmented reality libraries such as ARToolkitPlus, ARTag, Chilitags and AprilTag. Generally, these AR libraries come with fix number of predefined markers. In some AR applications, the number of required markers may be more

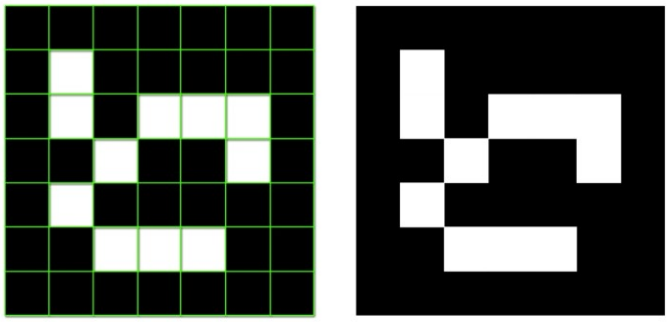


Figure 19. *ArUco marker (with ID 99).*

or less than the number of provided markers by AR libraries. In this case, using dictionaries with fix number of markers is not efficient. The ArUco allows users to create and use their own custom markers dictionary with the required number of markers and bits size [68]. Figure 19 displays a sample of customized marker using the marker generator tool provided by ArUco.

Most of the augmented reality and computer vision applications utilize the classic chessboard for camera calibration, but the ArUco uses its own calibration. According to the authors of the ArUco, their calibration board has one advantage over classic chessboard. This calibration board consists of several markers therefore, even if it is not visible completely, the camera calibration can be performed.

4.4 Google ARCore

In August 2017, Google presented an augmented reality Toolkit, called "ARCore", for building AR-based apps, games, etc. for Android devices [69], [70]. ARCore is a markerless AR SDK and does not require any extra devices or sensors than a smart phone has. There main functions that ARCore provides to programmers are divided into Motion Tracking, Environmental Understanding and Light Estimation.

Motion Tracking is the technology that makes the phone able to recognize its position in its surrounding world. When the phone is moving through the world, *Concurrent Odometry and Mapping* (COM) process is used by ARCore to find the position of the phone related to the world around. From the image captured by the camera, ARCore detects some point that are visually distinguishable in the image. These points are known as Feature Points which are used to compute the changing device location. Using the obtained information from the input image and data from the device's IMU, the pose of the camera is estimated in real-time.

Environmental Understanding: ARCore is able to detect flat horizontal surfaces, their size and location in the scene. ARCore uses Feature Points that are used in motion tracking, to understand the real environment more. It searches for the group of feature points that are placed on same horizontal surfaces, such as floor and tables, then ARCore represents them as planes to the user. Also, this feature can detect boundary of the planes that represent flat horizontal surfaces.

Light Estimation: This technology estimates the global lighting from the input camera image. It doesn't estimate position of point light source/s or the light direction in the captured scene. Actually, ARCore just estimates the global lighting by averaging the filtered intensity of pixel data of the current frame.

Besides these, ARCore provides some other features. One of these features is User Interaction that make users able to select or interact with synthetic object/s in the viewing environment. When the user taps on the screen, based on the hit point's screen space coordinates, a ray cast into the scene and it returns any hit object, plane or feature point with the pose of hit point in the word space.

The other feature is Anchors and Trackables. This feature helps to make sure the added virtual to the scene remains at the desire position even the device moves. Planes and feature points are known as Trackables. Because ARCore tracks these special objects continuously. In order to make sure that ARCore tracks the synthetics object in the scene, an anchor is needed to attach the virtual object to the trackables, based on returned pose from hit point.

4.5 Apple ARKit

ARKit (*Augmented Reality Kit*) is an AR API developed by Apple Inc and released on September 2017 for iOS 11. Due to having reasonable performance, it is required to use Apple A9 and later processors to perform ARKit. For world tracking, ARKit use *Visual-Inertial Odometry* (VIO) technology, which uses combination of input video and the motion sensor of the device. As with Feature Points in ARCore, ARKit finds notable features in the image of the scene and trace them based on changes of their positions and obtained information from motion sensors. ARKit is able to recognize flat horizontal planes in the captured scene, where user can add and place virtual objects on. In order to apply correct light intensity to virtual objects, ARKit use camera sensor to estimate the intensity of light in the device's surrounding environment.

There are some limitation and weakness that ARKit is suffering of. Since the tracking in ARKit is based on featured points that detected with analyzing the image, low quality of the input image may affect the accuracy of the tracking. Low light intensity and darkness in the scene affect tracking feature points. Beside analyzing image, device motion is used for tracking. Even a subtle movement makes ARKit able to have better understanding of

the captured environment [71]. On the other hand, extreme motion that makes image blurred, or very far distance for tracking, also reduce the tracking accuracy.

As a summary, Table 1 shows comparison of the mentioned frameworks in terms of license, platforms and type of tracking.

Table 1. *AR SDKs comparison.*

	License	Platform	Tracker
ARToolkit	GPL3	Cross-platforms	Marker-based /NFT
ARToolkitPlus	GPL3	Cross-platforms	Marker-based (ID Template /-Markers)
ArUco	BSD	Cross-platforms	Marker-based
Google ARCore	Apache	Android	Markerless
Apple ARKit	MIT	iOS	Markerless

5. THIRDEYE AR FRAMEWORK

As it is mentioned in Section 2.2, an AR application consists of fundamental parts that are common between all AR applications regardless of the application's type. Consequently, using a framework to handle the common parts could prevent redundancy and facilitates AR application development.

In order to develop AR applications on different platforms, the ThirdEye framework has been designed and developed. This Chapter introduces the ThirdEye framework. First, it gives an overview of the framework and briefly explain each part. Afterward, the technical aspects are discussed in detail for each desktop and mobile version of the framework in Sections 5.2 and 5.3.

5.1 Overview

ThirdEye is an object-oriented framework for augmented reality software development. It has been implemented in two versions for developing AR applications which targets desktop (Linux) and Android mobile platforms. It can be used to develop AR applications with different objectives.

ThirdEye framework consists of four main components, which are:

- Configuration manager
- Input manager
- Tracker
- Content generator (Renderer)

Each component is responsible for specific tasks that are independent from the other components and can be modified based on requirements of the application. Following sections explain these components briefly.

5.1.1 Configuration Manager

Each AR demo application that uses the ThirdEye framework may have its own configuration settings. The display settings, type of input data (camera, image file or video file), the tracker setting parameters etc., may be different from a demo to another. The parameters that are required for configuring different components of the program, can be set in the source code (hardcoding).

There is drawback for this practice. Every time that these parameters are needed to be changed, the source code should be modified and recompiled, which slows down the development process. This is especially more problematic in the earlier stage of development that usually the configuration of the application is required to be changed frequently.

So as to prevent recompiling source code after any changes in the settings, an external configuration file can be utilized. The configuration parameters, that are need for configuration, are stored in an external file. When the demo application starts up, these parameters are loaded and used for initialization of related components. ThirdEye uses this mechanism for configuration. This component is explained comprehensively in Section 5.2.3.

5.1.2 Input

Augmented reality applications rely on the input data that comes from real world. The input data in a visual AR application is visual data that can come from camera, video stream or an image file. In the ThirdEye framework, the input component responsible to extract visual data from desired input data source, and pass it to the related component. For example, tracker needs captured image from the real world so as to compute required data for tracking. In addition, content generator may need to use an image or images as texture for rendering. In Section 5.2.5 the input component is explained in more details.

5.1.3 Tracker

One of the most important component of an augmented reality application is the pose tracker. This component is responsible for calculating the camera pose and location of the virtual model in augmented environment based on specific features in the scene. In order to do so, it analyzes visual input data in real time using proper computer vision algorithms. The virtual world is defined and matched with real world based on this data. ThirdEye framework uses a cross-platform tracker library in both desktop and mobile versions of the framework. Section 5.2.4 discusses more about the tracker used in the framework.

5.1.4 Content Generator / Renderer

After pose tracker, the content generator is the second most important component of an augmented reality application. It generates augmented content that is required in the application. Based on the application objectives, it can be virtual 3D model, simple text or other type of visual content. Figure 20 shows two different types of generated content. Most of the augmented reality applications mainly use virtual 3D models as augmented contents. Generally, in computer graphics 3D model are generated using rasterization technique or other rendering techniques which are based on ray casting.

(a) (b)

Figure 20. *Results of two different type of content generator: a) 3D object b) simple line and text*

In an AR application that utilizes the ThirdEye framework, any rendering technique can be used for generating the content. Depending on the application objectives, the content generator may be a combination of different rendering techniques. The typical users of ThirdEye may replace the content generator component with their desired content generator to build their own application.

5.1.5 Workflow

Generally, the workflow of the AR application that uses ThirdEye framework, can be described with following steps:

- ***Initialization***
  The first step is initialization where all components are initialized based on the configuration data.

- ***Input***
  At the input step, an image data is extracted from desired source and then required visual data is prepared and passed to the relevant AR application components (e.g. Tracker or Renderer).

- ***Tracking / Pose estimation***
  The tracker searches and finds the marker in the input image data and performs pose estimation.

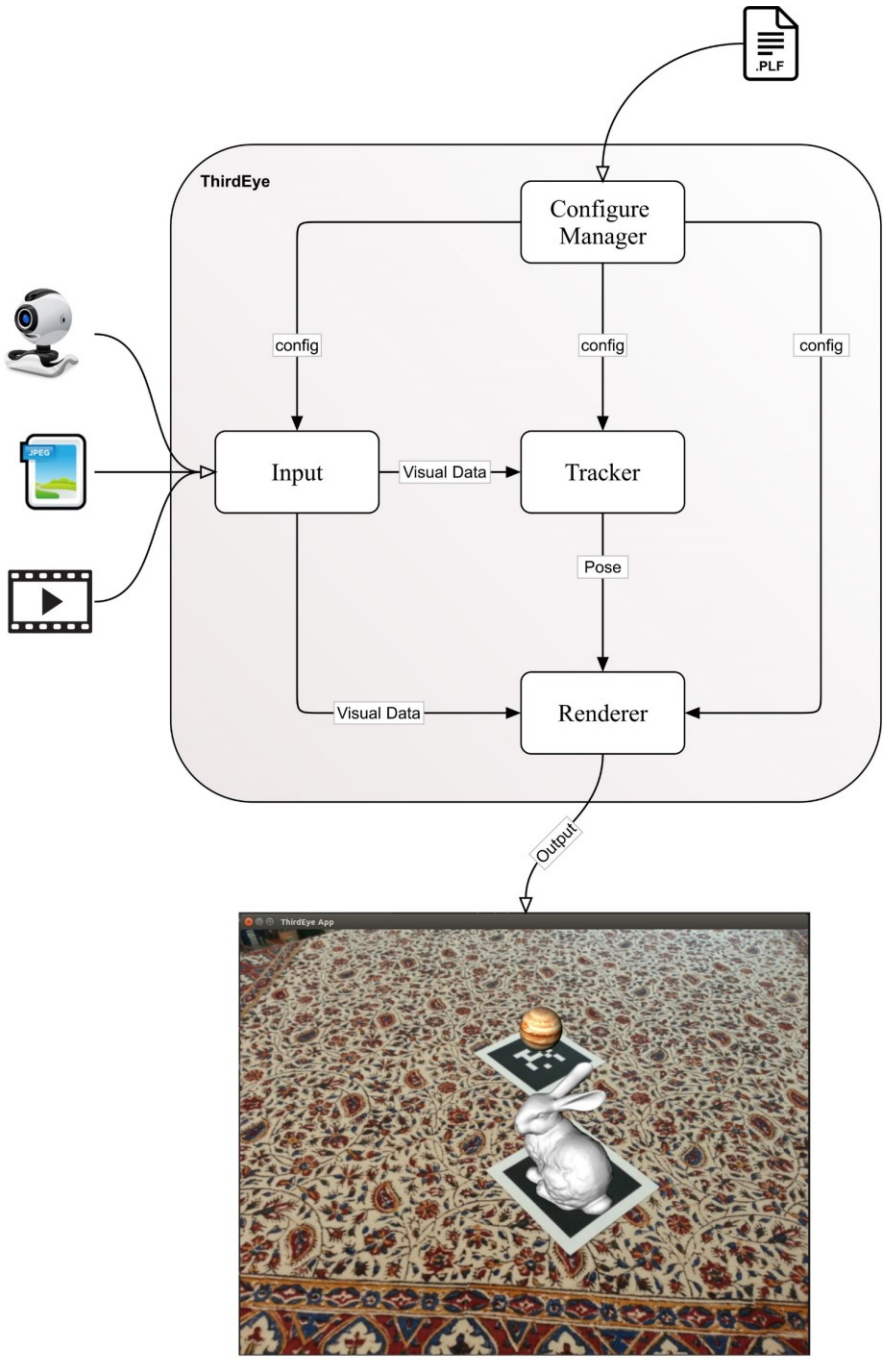


Figure 21. *ThirdEye workflow.*

- ***Content generation / Rendering***
  Based on the camera pose and the position of the marker in the scene, virtual content is generated and composite with the input image of the real world.

Figure 21 shows the overall workflow of the ThirdEye framework.

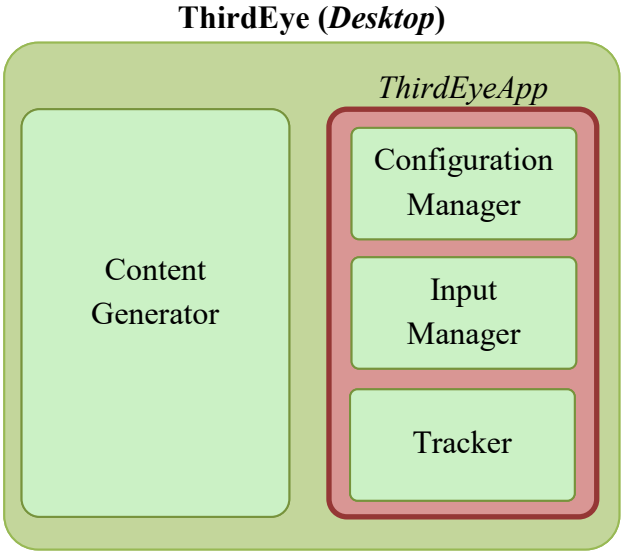


Figure 22. *ThirdEye framework - Desktop layout.*

5.2 ThirdEye – Desktop

Linux desktop platform is the first and main target of the ThirdEye framework. This section introduces the structure of the desktop version and discusses the implementation of its components. Additionally, camera calibration is discussed in Section 5.2.9 and it is explained how to use ThirdEye to develop an AR application on desktop in Section 5.2.10.

5.2.1 Structure

Simplicity has been one of the main objective for designing the structure of the ThirdEye framework. In augmented reality research, where usually the research area focuses on particular part of the augmented reality, some components are constant and common between every AR applications during the research process. In case of research on the rendering in augmented reality, except content generator that may be substituted frequently, the other components remain the same. In order to achieve simplicity in the ThirdEye framework, the common components are unified as a single component. The configuration manager, input manager and tracker are the common components that are included in the ThirdEyeApp class. This class is described precisely in Section 5.2.6. It is obvious that the common components can be used separately. The purpose of unifying them is simplification and facilitating the development process. Figure 22 illustrates the structure of the desktop version of the framework.

This framework is being developed using object oriented programming style in order to achieve reusability, maintainability and convenient software design. The software architecture of the ThirdEye framework is designed in a way to keep the balance between

performance, maintainability and simplicity. It is implemented in C++ 14 to get benefit from the modern C++ language features.

5.2.2 External libraries

The ThirdEye framework utilizes several external libraries. Some of these libraries are included in the framework source code, but some of them must be obtained separately and installed. These external libraries are listed in this section, along with some brief description.

- ***ARToolkitPlus*:**
  An open-source marker-based pose tracker library. This library is cross-platform which allow us to use it in both desktop and mobile version of the framework. In Section 4.2 this library is explained in more details.

- ***OpenGL*:**
  A cross-platform graphical API which is used for creating 2D or 3D interactive application. The desktop version of the framework requires the OpenGL 3.3 or above. Generally, the latest version of the OpenGL is provided by the graphics card vendors or open-source libraries.

- ***GLFW*:**
  an open-source and cross-platform graphic framework that facilitates using the OpenGL API for developing 2D/3D graphics application. This library is used in the ThirdEye framework for initializing OpenGL and handling keyboard and muse input.

- ***OpenCV*:**
  a multi-platform library that provides a rich set of functionalities and tools useful for computer vision and image processing. The OpenCV is used in this framework for handling input image data and image processing.

- ***GLM*:**
  a mathematics library that implements GLSL specifications in C++ language. This library provides mathematical functionalities that are usually required in a computer graphics software.

- ***AntTweakBar:***
  an open-source library that has been implemented for creating graphical UI for graphic application.

The GLM, AntTwaeakBar and ARToolkitPlus are included in the framework source code as third-party libraries (in 3rdparty folder)

5.2.3 Configuration Management

In initial version of the framework, the configuration parameters were stored in a XML file. When the demo started up, the parameters were extracted from the XML file using a third-party library, TinyXML. Program 2 shows part of the XML file that was used to store the configuration data.

```
<ThirdEyeConfig>

    <windowTitle>ThirdEye Sample Application</windowTitle>

    <screen>
        <width>1024</width>
        <height>768</height>
    </screen>

    <input>
        <mode>image</mode>
        <cameraIndex>0</cameraIndex>
        <size>
            <width>1920</width>
            <height>1080</height>
        <size>
        <videoFile>videos/test_video01.mp4</videoFile>
        <imagesList>
            <image>images/test_image01.jpg</image>
            <image>images/test_image02.jpg</image>
            <image>images/test_image03.jpg</image>
        </imagesList>
    </input>
      .
      .
      .
</ThirdEyeConfig>
```

Program 2. *Configuration file using XML format.*

Later, it was decided to replace the TinyXML library by more lightweight code and use a simpler text format for storing configuration parameters instead of XML. Two goals have been pursued by this decision:

- Lessening dependency of the framework on external libraries.
- Making configuration data format simpler and clear.

The *ParametersList* is a template-based class that has been implemented for handling configuration data in the ThirdEye framework. It provides a container, called *parameters*

list, which is used for storing and manipulating the parameters that are required for configuration. The parameters and their values can be added to the parameters list inside the source code, or directly from an external file. The parameters are stored in a text file format with extension PLF (*Parameters List File*) in a specific syntax. Program 3 shows a simple ParametersList syntax that presents the same data in Program 2.

```
windowTitle = ThirdEye Sample Application;
screen =
{
    width = 1280;
    height = 720;
    isFullScreen = false;
}

input =
{
    mode = video;
    cameraIndex = 0;

    size =
    {
        width = 1920;
        height = 1080;
    }

    videoFile = videos/video_01.mp4;
    imageList = image_01.jpg, image_02.jpg, image_fhd_03.jpg;
}
.
.
.
```

Program 3. *Configuration file using PLF format.*

The syntax which is used to store the data, has been designed in a way that makes it readable and easy to parse. Some simple rules have been defined that should be followed:

- Parameter and its value are separated by '='.
- The value of a parameter can be numerical, Boolean or String.
- Semicolon ';' should be placed at the end of a statement.
- '#' is used for commenting a single line.
- '{' and '}' are used to create a group of related parameters.

In plf file, the parameters that are logically related can be grouped together in order to increase readability. A group is defined same as a normal parameter, but instead of referring to a single value, it refers to the parameters that are part of the group. The members

of a group are placed in a block that is defined by curly braces '{}'. The ParametersList supports nested group feature, so a group itself, can have other groups as its member.

The following (Program 4) demonstrates how the ParametersList class is used for parsing plf file, extracting, storing and manipulating the parameters.

```
ParametersList pl("config01.plf");

std::string title = pl.get<std::string>("title");
int width = pl.get<int>("Screen.Size.width");

pl.set("v-sync", true); //v-sync is not in the list so it will be added.
pl.set("Screen.isFullScreen", true); //Updates isFullScreen value
```

Program 4. *ParameteraList object usage.*

A plf file can be loaded and parsed when a ParametersList object is created or by calling the *loadFile()* function. When a configuration file is loaded, all specified parameters and their values are stored in the parameters list. If another plf file is loaded, the already existing parameters in the list will be updated and the new ones will be added. In the runtime new parameters can be added or existing ones can be altered using *set (parameter_name, value)* function. First argument is a string that refers to name of the parameter and second one is its value. In order to access a sub-parameter, a combination of the group name and the sub-parameter is used that are separated by '.' character. For example, "Screen.isFullScreen". The *get<T>()* function is used for retrieving the value of a parameter. T is the type of the value that we want to get. It can be a C++ build-in type or a string.

In the ThirdEye framework, an object of the ParametersList class is created and used for storing the parameters that are required for configuring different components of the application. In Section 6.1.3, it has been explained that how ParametersList is used to configure a ThirdEye application.

5.2.4 Tracker

Camera pose estimation (calculating position and orientation of camera) and computing the position of the virtual object in the scene are done by the tracker component. It has huge impact on performance and quality of the application. Currently there are many different pose trackers with various capabilities and features that are suitable for different purposes. Before starting to design and implement the framework, a research was done about available pose tracker libraries and SDKs to find a proper one which provides functionalities that are required in the framework. During the research, some criteria were considered.

- ***Free and open-source***:

  There are some commercial and closed-source augmented reality SDKs that offer a wide range of functionalities such as marker-less and 3D object tracking that are useful for creating high quality AR applications. Based on the thesis's objectives and requirements, the focus has been on available free and open-source libraries. They provide accurate pose estimation which is sufficient for this master thesis work. In addition, we have more control over open-source code. It can be modified if needed, in order to gain better performance or add more features.

- ***Cross-platform***:

  The ThirdEye framework was planned to be implemented in two versions, for PC and Android devices. It has been preferable to use same AR library for both versions, because less code modification is required. So, the chosen AR SDK for this framework should support these two platforms.

- ***Lightweight***:

  This factor specially is important for mobile devices which have limited hardware.

After the initial search, ARToolkit, ARToolkitPlus and ArUco were considered for further research. After more study and comparison between these four libraries, the ARToolkitPlus was chosen for the ThirdEye framework. It is an open-source marker-based pose tracker library, which satisfies the criteria that have been considered for choosing a proper library for the framework. In contrast to other three SDKs, the ARToolkitPlus does not depend on any external library which allows it to be ported to different platforms easily. Furthermore, it has been optimized for mobile devices that makes it better candidate for mobile version of the framework. ARToolkitPlus is explained in more detail in Section 4.2.

Regardless of which pose tracker libraries or SDK that is used, camera calibration is one of the important initial process that must be done beforehand of tracking process. For each camera, the calibration needs to be done once. There are different methods and tools for camera calibration. For this thesis work, a simple OpenCV tool has been utilized for camera calibration. In Section 5.2.9, it is explained how it can be used for this purpose.

In the ThirdEye framework the *ARTPTracker* class is used for Pose estimation and tracking markers in the scene. This class utilized ARToolkitPlus for this purpose. The parameters that are required for configuring the tracker are placed in the *tracker* group of the configuration file (Program 5).

```
tracker =
{
    calibrationFile = camera_data.xml;
    patternWidth = 2.0;
    threshold = 158;
    borderWidth = 0.25;
}
```

Program 5. *Tracker configuration parameters in plf file.*

5.2.5 Input / Input Manager

The visual trackers process visual input data to detect a specific feature which can be used for camera pose estimation. Some of the AR SDKs and tools have been integrated with external libraries, which supply functionalities that are required for capturing and handling visual data. As it has been mention before, the ARToolkitPlus is only a pose tracker library. It does not provide any mechanism for capturing visual data that is required for pose estimation.

In the desktop version of ThirdEye framework, the OpenCV version 3.0+ library has been used for handling visual input data. We have chosen the OpenCV for following reasons:

- It allows us to capture image data from camera and use video and image file with different file format in the framework.
- It is cross-platform and makes porting the ThirdEye framework to another operating system and platform possible.
- It provides a rich set of functionalities for advance image processing and manipulation. The OpenCV can help to analyze visual input data further to get better understanding of the scene. Additional information from the scene is useful to create more accurate augmented content in the AR application. For example, it can be used for estimating the light source direction to generate more precise shadow effect.

In the ThirdEye framework, the InputManager is responsible for handling visual data that is needed for pose estimation and content generation. This class facilitates accessing camera, video or image file via the OpenCV library and provides additional features, which are useful for managing input data. The *InputManager* class is inherited from the *Input* class. The *InputMode* enumerator is used for specifying source of the input data which is image, video stream or camera (Figure 23).

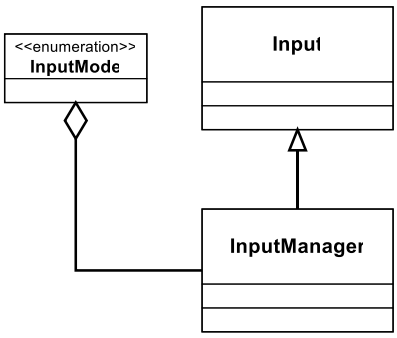


Figure 23. *InputManager class diagram.*

In the configuration file, the parameters that are used for configuring the InputManager, are placed in the *input* group (Program 6).

```
input =
{
    mode = camera;
    cameraIndex = 0;
    size =
    {
        width = 1920;
        height = 1080;
    }
    videoFile = video_fhd_01.mp4;
    imageList = an_image.jpg, image_hd.png, image_fhd.jpg;
}
```

Program 6. *Input manager configuration parameters in plf file.*

In this group, *mode* refers to the type of the input source that the visual data comes from. Its value can be camera, video or image. Based on the input mode, other relevant parameters are used. If the mode is camera and there is more than one camera in the AR system, *cameraIndex* indicates which one should be utilized. The width and height of the image data, which are specified in *size* group, are needed for initializing the tracker and the camera. The input video and image files are determined by *videoFile* and *ImageList* parameters. If more than one image file is specified, it is possible to switch between them during the run-time via InputManager. This is especially useful when the user wants to examine the generated content for different images, without rerunning the application for each image. If these parameters are not specified by the user the default value are used. Camera with index 0 is the default input source.

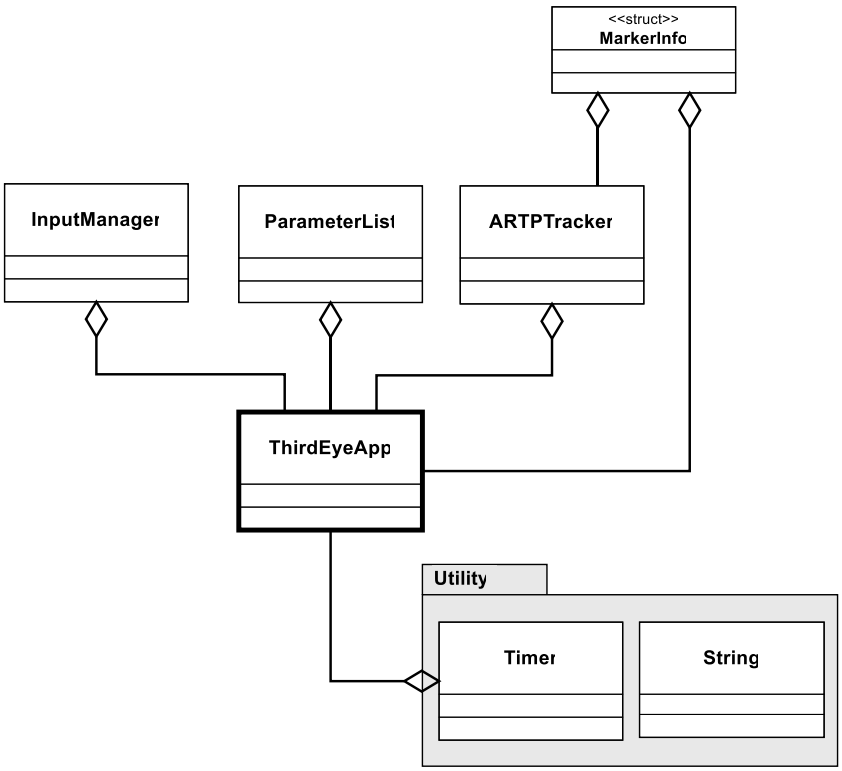


Figure 24. *ThirdEyeApp class diagram.*

5.2.6 ThirdEyeApp

The ThirdEyeApp class is the base of each application that uses the ThirdEye framework. It contains the input manager and the tracker components that are required by every augmented reality application, despite the type of content generator. This class uses an ParametersList objects to keep configuration parameters. When a ThirdEye application starts, the input manager and the tracker components are created and configured based on configuration parameters that are provided by the user. An AR application may use rasterization rendering or any other rendering technique to generate augmented content, therefore the content generator is not part of the ThirdEyeApp class. Figure 24 presents the class diagram of ThirdEyeApp for desktop version.

5.2.7 Content Generator / Renderer

ThirdEye has been implemented in a way that makes it possible to work with either techniques. In this framework, the OpenGL library is utilized for rasterization rendering. The ThirdEye framework provides some basic functionalities that facilitate using OpenGL 3.3 and above, such as:

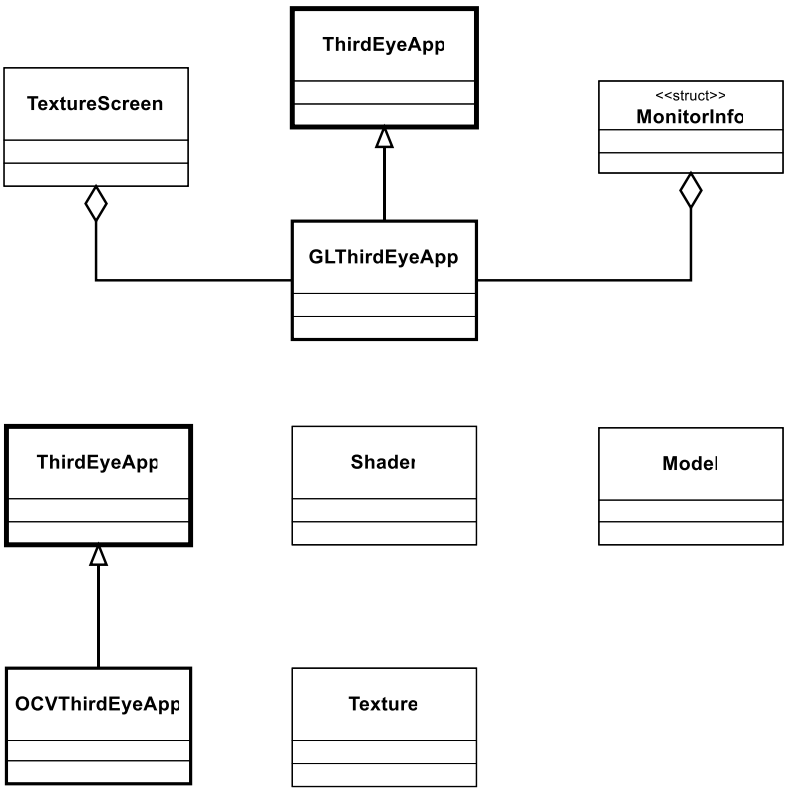


Figure 25. *GLThirdEyeApp and OCVThirdEyeApp class diagrams with the related classes.*

- ***GLThirdEyeApp***:

  This class serves as the base of every ThirdEye application that uses the OpenGL library for rendering.

- ***Shader***:

  The Shader class provides functionalities which simplify using the GLSL shader programs in a ThirdEye application.

- ***Texture***:

  This class is responsible for loading, creating and using textures in the AR application. For loading image files, the functionality that provided by the OpenCV is utilized.

- ***TextureScreen***:

  TextureScreen is used for rendering single texture. Generally, this class is employed for rendering input image data in this framework.

- ***Model***:

  The Model is responsible for loading 3D model data from OBJ file and preparing it for rendering.

In the ThirdEye framework, using other rendering techniques for generating augmented content is also possible. Usually the result of a rendering process is presented as image data. If the GLThirdEyeApp is used for creating the application, this image can be showed by the TextureScreen. Also, the functionalities provided by OpenCV library can be utilized to display the result of the rendering process. In some ThirdEye demos, using the OpenGL is not necessary, for example simple text, label or line. In this case, *OCVThirdEyeApp* can be used instead of GLThirdEyeApp. Like other components of the framework, content generator also is configured by configuration manager. Figure 25 displays the GLThirdEyeApp and OCVThirdEyeApp class diagram with relevant classes to the content generator.

5.2.8 Assumptions

In order to make sure implemented AR application using ThirdEye functions properly, there are some assumptions that should be considered. These assumptions are listed and the reasons why they must be considered are explained, bellow.

- ***Calibrated Camera*:**
  Without calibrating camera, the tracker is not able to perform pose estimation accurately. Therefore, virtual content may be rendered at incorrect position.

- ***Using predefined fiducial markers*:**
  In case of using marker-based tracker, likewise ARToolkitPlus that is employed in ThirdEye, tracker cannot detect marker if it is not predefined for the tracker. Accordingly, the AR application doesn't work.

- ***The captured scene contains at least a marker*:**
  Without marker in the scene, marker-based tracker is not able to perform tracking and pose estimation.

- ***Marker must be completely visible*:**
  The ARToolkitPlus uses square markers. For pose estimation all four corners of the marker should be visible. Even if a small portion of the marker be covered, invisible or out of the input image, tracker is not able to recognize it.

- ***Correct modeled virtual object*:**
  In case of using 3D objects, it is assumed the object file has correct vertices positions, normal vectors and face indices data. Incorrect data causes failure in rendering or light effects calculation etc.

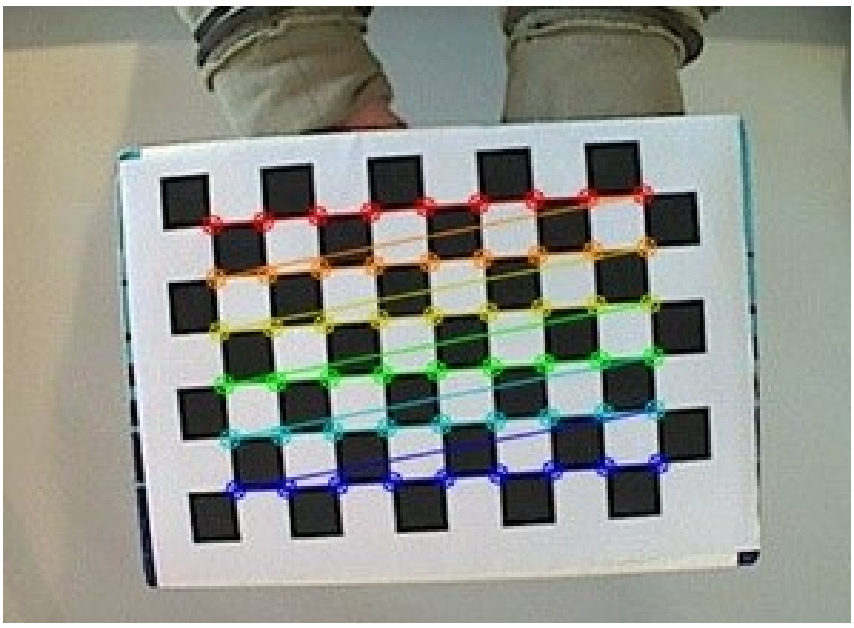


Figure 26. *Camera calibration using chessboard pattern [72].*

- ***The brightness of the captured scene must be in normal range*:**
  Too dark or too bright input image may affect the marker detection and pose estimation.

These assumptions are associated with this work and those may be changed based on customized components and different application.

5.2.9 Camera Calibration

It is assumed that the camera calibration is done beforehand so as to maximize pose estimation accuracy. For this work, an open-source OpenCV camera calibration tool is utilized [72]. This tool uses sequential captured images of a chessboard pattern which has a size of 9 x 6 to estimate parameters of the camera (Figure 26). These parameters are camera intrinsic, camera extrinsic, and distortion coefficients. Besides the using sequential image, the input can be from camera stream output. Calibration is required to be done once for per camera. Once the Camera calibration is done, the result is saved into an OpenCV style XML for further use with tracker.

5.2.10 How to Use ThirdEye Framework

In this section, using an example, it is explained briefly how the ThirdEye framework can be used to create a simple AR demo. This framework has been designed in object-oriented way. Each of its main components can be extended and customized based on the application objectives. For example, the GLThirdEyeApp is the base class for any AR demo

which wants to leverage OpenGL library. It sets up OpenGL, implements main loop of the application and handles keyboard and mouse input. The GLThirdEyeApp, itself is inherited from the ThirdEyeApp class. This class sets up the tracker and the InputManager components.

In order to create an AR demo, a class is created and derived from the GLThirdEyeApp class (for example *ARSimpleApp* class in Program 7).

```
class ARSimpleApp : public ThirdEye::GLThirdEyeApp
{

 public:
      ARSimpleApp(void);
      virtual ~ARSimpleApp(void);


      void initialize(void) final;
      void render(double elapsedTime) final;
}
```

Program 7. *A simple AR demo class.*

In the *initialize* function (Program 8), main components of the application such as tracker and the InputManager are configured based on the configuration settings. The derived class (in this case ARSimpleApp) has access to an object of ParametersList, named mConfigParameters which keeps all parameters that are needed for configuration. The default values of the parameters are set by the ThirdEyeApp and GLThirdEyeApp classes. The printAll function of the mConfigParameters can be called to see all default values that are used for configuration. In the initialize method, if it is needed, these values can be modified by loading a configuration file (.plf file) or via *set* function of the mConfigParameters. It is necessary to call base class initialize function for initialization. In initialization process, the tracker (mTracker) and the InputManager (mInputManager) objects are crated and accessible for the derived class.

```
void ARSimpleApp::initialize(void)
{

      mConfigParameters.loadFile("ARSimpleApp.plf");

      mConfigParameters.printAll();

      GLThirdEyeApp::initialize();

      //We can limit the frame rate based on the input video FPS.
      if(mInputManager->getInputMode() == InputMode::Video)
            setFPSLimit(mInputManager->getVideoFPS());
}
```

Program 8. *Initialize function.*

The *render* is the main loop function that is invoked for each frame (Program 10). This is pure virtual function that should be implemented by the derived class. Rendering of the visual entities is done in this function. The *elapsedTime* parameter that is passed to this function indicates the amount of time that it takes to complete a frame. Also, the current frame rate can be obtained by *getFPS* function. It is possible to limit frame rate of the application via *setFPSLimit* function. This is especially useful when it is necessary to adjust rendering speed of the application with the camera or input video actual frame rate.

Positions of the virtual entities (3D model or text) in the augmented environment depends on the location of the detected marker in the scene. The input image data is analyzed by the tracker for pose estimation and calculating the location of the marker. This should be done in the render function before rendering the augmented content. Firstly, the image data which is needed by tracker, is retrieved via *getFrame* function of mInputManager. The InputManager captures image data from the source (camera, video or image file) that is specified during initialization.

Now the image data can be passed to the tracker for locating the markers in the scene. The findMarkers function of *mTraker* object returns a list of the detected markers' information in the scene. The MarkerInfoList is a STL vector of MarkerInfo structure that stores the detected marker's information such as marker id, model matrix and center point Program 9). If the desired marker is found, its model matrix or center point can be used for placing 3D or 2D entity in the augmented environment.

```
struct MarkerInfo
{
      ui16        id;
      glm::mat4   modelMatrix;
      Point       center;
}
```

Program 9. *MarkerInfo struct.*

In the ThirdEye framework OpenGL can be utilized for rasterization rendering. It is already set up by GLThirdEyeApp and some basic functionalities have been provided which facilitate the use of OpenGL. However, it is also possible to use other rendering technique, for example ray tracing. It is just needed to render the virtual models using chosen technique and draw the result on the input image data. Then by using the OpenGL, the image data is rendered on the output screen. In this framework, the TextureScreen class is responsible for rending image data. When a GLThirdEyeApp is initialized, an object of the TextureScreen is created (mTextureScreen) that is accessible by the derived class.

In the ARSimpleApp demo, the OpenGL or other rendering technique has not been used for creating augmented content. In this demo, the simple drawing functionality provided by the OpenCV library is used to draw line and text on the detected markers in the scene.

In the render function (Program 10), after locating markers, a text that indicates the detected marker id appears on top of it. If more than one markers are detected, a line is drawn from center of one marker to the next one.

```
void ARSimpleApp::render(double elapsedTime)
{
 //Getting input image.
 Frame inputFrame = mInputManager->getFrame();

 //Tracking the markers in the input image and storing
 //the found markers in the MarkerInfoList list.
 MarkerInfoList foundMarkers = mTracker->findMarkers(inputFrame.data);

 //Going through the found markers.
 for(i32 iMarker = 0; iMarker < foundMarkers.size(); ++iMarker)
 {
      if(iMarker > 0)
      {
            cv::line(inputFrame,
            foundMarkers[iMarker - 1].center,
            foundMarkers[iMarker].center,
            cv::Scalar(255, 0, 0, 255),
            2);
      }

      //Updates the label text.
      mMarkerLabel->setText("ID : " +
                        std::to_string(foundMarkers[iMarker].id));

      mMarkerLabel->setPosition(foundMarkers[iMarker].center);

      //Draws on the image.
        mMarkerLabel->draw(inputFrame, true);
 }

 //Render input image.
 mTextureScreen->render(&inputFrame);
}
```

Program 10. *Render function.*

After creating our demo class, it can be instantiated and run it as the Program 11 shows.

```
int main(int argc, char* argv[])
{
    ARSimpleApp arDemo;
    arDemo.initialize();
    arDemo.run();

    return 0;
}
```

Program 11. *Main function.*

Figure 27 shows the output of the ARSimpleApp demo which contains two markers where the center of the markers are labeled and connected to each other via a line.

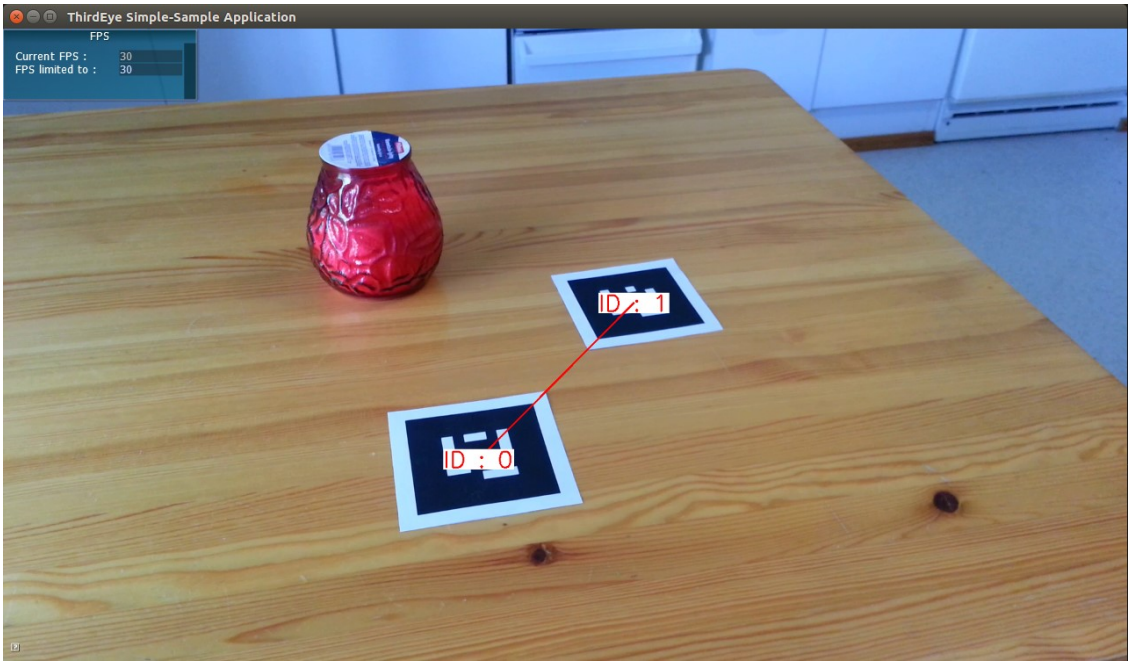


Figure 27. *ARSimpleApp demo output with two markers connected by line.*

5.3 ThirdEye – Mobile

The usage of augmented reality in mobile based applications is clearly increasing in different fields such as education, entertainment and business. The mobile devices become more powerful that makes it possible to use them for more hardware-demanding applications. On the other hand, the mobility characteristic of these devices makes them more practical and convenient for AR applications. In this work, beside the desktop version, a mobile version of the ThirdEye framework has been implemented for the Android devices.

The mobile devices have different hardware and software characteristics compared to the desktop computers. They have less hardware resources (processor power, storage, etc.) and use different operating system. So, the mobile version of the ThirdEye framework is implemented slightly different based on these differences.

This section discusses the mobile version of the ThirdEye framework, its similarities and differences with the desktop version, the structure of this version of the framework and its major component.

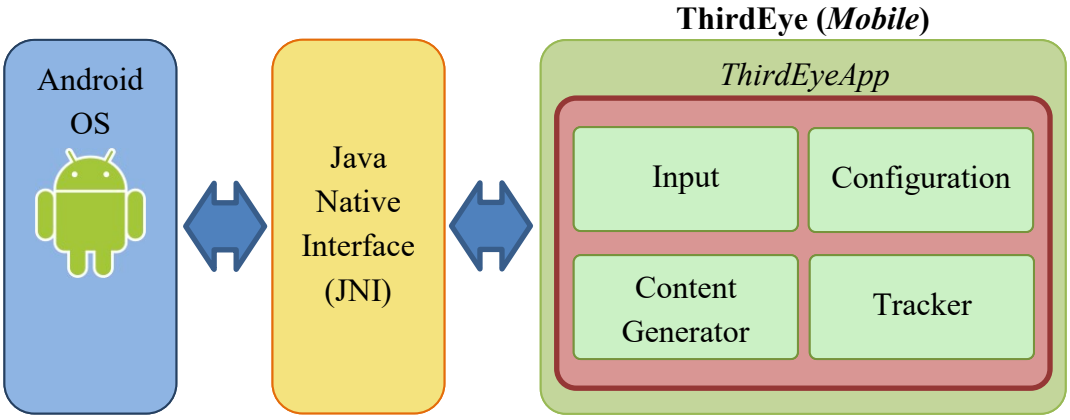


Figure 28. *ThirdEye framework - Mobile layout.*

5.3.1 Structure

Java is the main programing language that is used for developing software for Android devices. Android SDK provides a rich set of tools and functionalities for Android application development. The desktop version of the ThirdEye framework has been implemented in modern C++ (C++ 14). If the Android SDK were used to implement the mobile version of the framework, the source code should be rewritten in Java language which was time consuming and unpractical. Also, there was a negative impact on performance.

Fortunately, the *Android NDK* (Native Development Kit) allows programmers to use C/C++ to implement their applications for Android devices. This makes it easier to port the applications to Android that have been already written in C/C++ for another platform. Also, as the source code is compiled to machine code instead of Java bytecode, it yields better performance.

The core of the mobile version of the ThirdEye framework, like the desktop version, has been implemented in modern C++ (C++ 14). One of the differences between the structure of these two versions is the ThirdEyeApp class. In the desktop version the Input, Configuration and Tracker are wrapped in the ThirdEyeApp class. The Content Generator component is separated, thus the user of ThirdEye can replace it with other content generators. In contrast, in the mobile version the Content Generator is also included in the ThirdEyeApp class (Figure 28).

In the mobile version of the ThirdEye framework, some part of the framework which is responsible for handling Camera and UI has been implemented in Java language. The *Java Native Interface* (JNI) is used as the bridge between Java code and the native code. Figure 28 shows the structure of the framework on mobile version.

5.3.2 Assumptions

Beside the mentioned assumptions in Section 5.2.8, following assumptions should be considered for using and testing the mobile version of the ThirdEye framework as well.

- Integrated camera
- Android 4.4 KitKat (API 19) and above
- OpenGL ES 2.0

5.3.3 Comparison of Two Versions

There are some differences between the desktop and mobile version of the ThirdEye framework. Most of the classes, such as Shader, Texture, Model, Buffer, and ARTPTracker remain the same as the desktop version, except some minor modifications that were required to applied. In this section these differences have been discussed.

- ***Camera calibration*:**
  In the desktop version, the parameters that are required by the tracker, for camera calibration, are extracted from an external file. In the mobile version, a set of generic parameters are used for calibration. These parameters have been set in the tracker source code.

- ***Content generator/Renderer*:**
  In the desktop version, the OpenGL 3.3+ has been utilized for rasterization rendering. However, the OpenGL ES 2.0 is used for this purpose in the mobile version. Hence, some part of the code was modified so as to have compatibility with the OpenGL ES 2.0.

- ***Input*:**
  In desktop version the visual input data can come from camera, video stream or an image. In contrast, the mobile device camera is the only visual input source in the mobile version.

- ***Configuration*:**
  In Android SDK there are different data storage techniques available. One way to store data in an Android app is *Shared Preferences*, where primitive data can be saved in key-value pairs. The Android platform stores an app's Shared Preferences in an xml file internally in a private directory. The mobile version of the ThirdEye framework uses this method instead of using plf file. Also, configuration can be done using setting property via the mobile application UI (Figure 29).

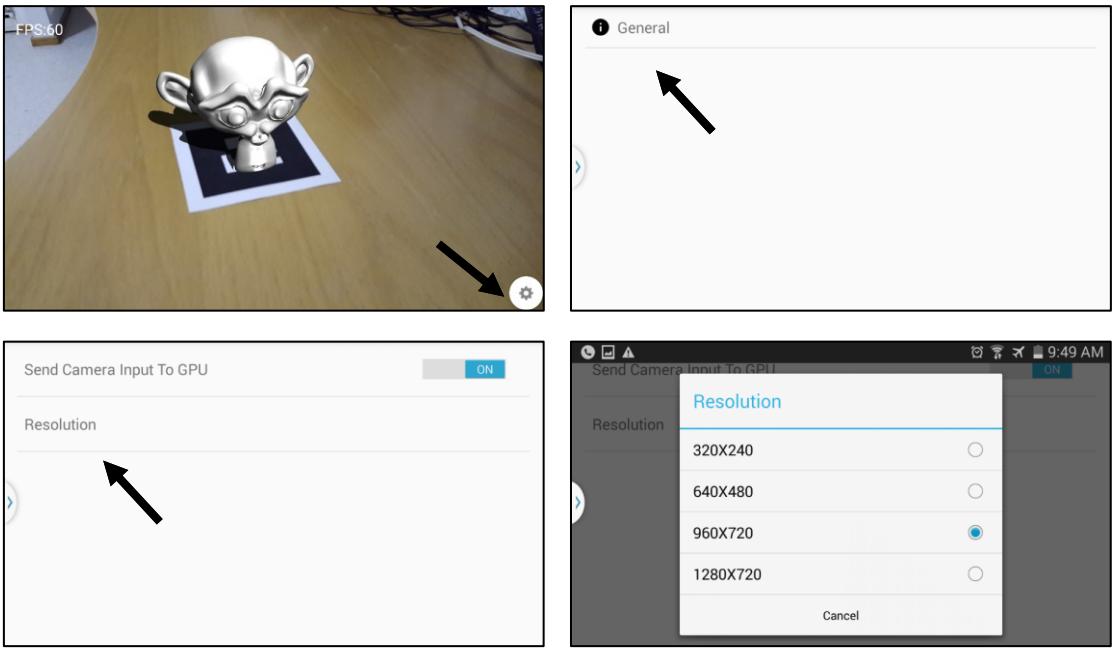


Figure 29. *Using the mobile app setting for configuration.*

- ***UI***:
  In the desktop version a third-party library called *AntTweakBar* is used in order to create a simple graphical UI. In the mobile AR demos, the Android built-in graphical UI components are utilized for this purpose.

- ***TextureScreen class***:
  In the mobile version, the format of the capture image is YUV. To convert this format to the RGB, before sending it to the GPU, some changes were applied to this class.

- ***Exception handling***:
  In desktop version, C++ standard exception is used for handling run-time errors. For the mobile version, return codes are used to indicate whether a method/function is executed successfully or not.

5.3.4 Color Conversion

As it is mentioned in previous section, the camera of the mobile device is used to capture the real-world image. The default color space that Android devices use for camera output image is YUV-NV21. In this format, the image data is divided in two planes. The first one with the size of (width x height), is Y channel which represent luminance (one byte per pixel) and the second plane with the size of (width x height)/2 bytes keeps the UV values that depict color information of the image. Therefore, for every 2 by 2 pixels group

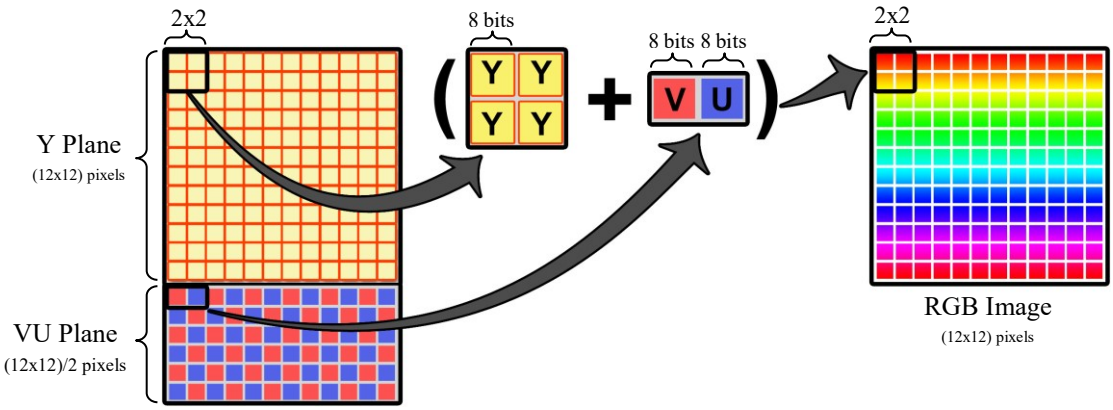


Figure 30. *YUV-NV21 color space (adapted from [73]).*

there are 4 samples of Y, one sample of U and one sample of V. The size of the image data in YUV-NV21 equals to (width x height) x 3/2 bytes. The YUV-NV21 and YUV-NV12 are the same but in contrast to the YUV-NV12, in YUV-NV21, V component comes first, followed by U. Figure 30 illustrates the YUV-NV21 color space for an image with size of 12 x 12 pixels.

In the mobile version Of the ThirdEye framework, OpenGL ES is used for rendering the input image data on the on the screen. In the OpenGL image data is stored in RGB format. Hence, before rendering input image, it is required to convert its color format from YUV-NV21 to RGB. This color-space transformation can be done in CPU or GPU (using the fragment shader).

In the initial version, the color format conversion was done in CPU. After capturing new frame from camera, its format was changed to RGB and uploaded to GPU for rendering. This process was time consuming and reduced overall performance of the application. Most of the mobile devices that are available today, are equipped with multi-core processors. This feature can be utilized for speeding up the programs that are parallelizable. The color conversion in this framework also can benefit from this feature. The color of each pixel is computed independently from other pixels therefore it can be parallelized easily. The desktop and mobile version of the framework has been implemented in the standard modern C++ (C++14) and fortunately, it supports multithreaded programming via its build-in standard library. Thus, it is not required to use any external library for this purpose.

When the mobile demo starts, the number of supported hardware concurrent threads on the device is obtained (n). During the color conversion, n - 1 separate threads are created and each one processes specific part of the input image. The performance of the multi-thread version of the color conversion is evaluated and compared with the single-thread version in Section 6.3.2.

Figure 31. *Output image tearing artifact.*

Although, multithreading color conversion increases the overall performance of the application, it causes a visual artifact on the output image. When the mobile device is moved fast, the output image is torn in the parts of the image that is divided for the threads. This artifact is shown in Figure 31. In this example, three threads were used for color conversion.

Later in the development process, it was decided to utilize GPU of the mobile device instead of CPU, for changing the input image format. Generally, the GLSL shader program is used in order to employ GPU for color conversion in the Android and iOS platforms. The same technique has been utilized for this purpose in the ThirdEye framework. Like desktop version of the framework, the TextureScreen class in the mobile version is responsible for rendering input image data. In order to make it enable to render YUV image date, it was required to apply some changes in it.

In OpenGL, image data is stored in a three or four-channel texture object, in RGB format. Each channel keeps a color component (red, green, blue or alpha) of the image. Previously the TextureScreen used a GL_RGB texture object to store input image data. As it has been mentioned before, in YUV format, brightness (Y) and color components (UV) are stored in two separate planes. The sizes of these planes are different so it is not practical to keep all the image data using a single texture object. In the TextureScreen class, two texture objects are created and used to store the image data. One single-channel texture (GL_LUMINANCE) for Y component and one two-channel (GL_LUMINANCE_ALPHA) texture for UV components. In the render function of the TextureScreen, it is checked to see whether a new input image data is received or not. If new input image is available, its Y and UV components are sent to GPU texture memory, separately. Then

in the fragment shader, the Y, U and V components that are required for calculating red, green and blue values of the current fragment, can be extracted from GPU texture memory.

```
1  float y = texture2D(textureSamplerY, fragUV).r;
   float v = texture2D(textureSamplerUV, fragUV).r;
3  float u = texture2D(textureSamplerUV, fragUV).a;
```

Program 12. *Extracting YUV in fragment shader.*

The textureSamplerY and textureSamplerUV samplers represent textures that contain Y and UV values of the image (Program 12). The *texture2D* function samples the given textures at coordinates that specified by fragUV. The luma (Y) is stored in R channel. The color components (V and U) are kept in R and A channels of the UV texture, respectively. The value of Y is between 0 and 1 where the U and V components have a value between -0.5 and 0.5. After extracting luma and color components, RGB value of the pixel can be computed. By doing color-space transformation on GPU, workload of the device CPU can be reduced and the overall performance of the application be increased.

The performance of the GPU and CPU versions of the color conversion are compared in Section 6.3.2.

6. EVALUATION

There are two versions of ThirdEye framework for AR application development. The desktop version is built on Linux and the mobile version targets Android platform. In order to evaluate and demonstrate usage of the framework, some AR demo applications are implemented. These demos include simple sample AR application that is described in Section 5.2.10, and implementation of some of light illumination effects such as shadow and refraction in augmented reality.

This Chapter discusses the comparison between the ThirdEye framework and the related works that are mentioned in Chapter 4. Further, Sections 6.1 and 6.2 describe the application of the ThirdEye framework on both desktop and Android mobile platforms. In Section 6.3, the implemented demos are benchmarked in terms of performance. The used hardware for benchmarking is mentioned in Section 6.3.1. The color conversion in mobile version is also benchmarked for single and multiple thread on three different mobile devices, in the same section.

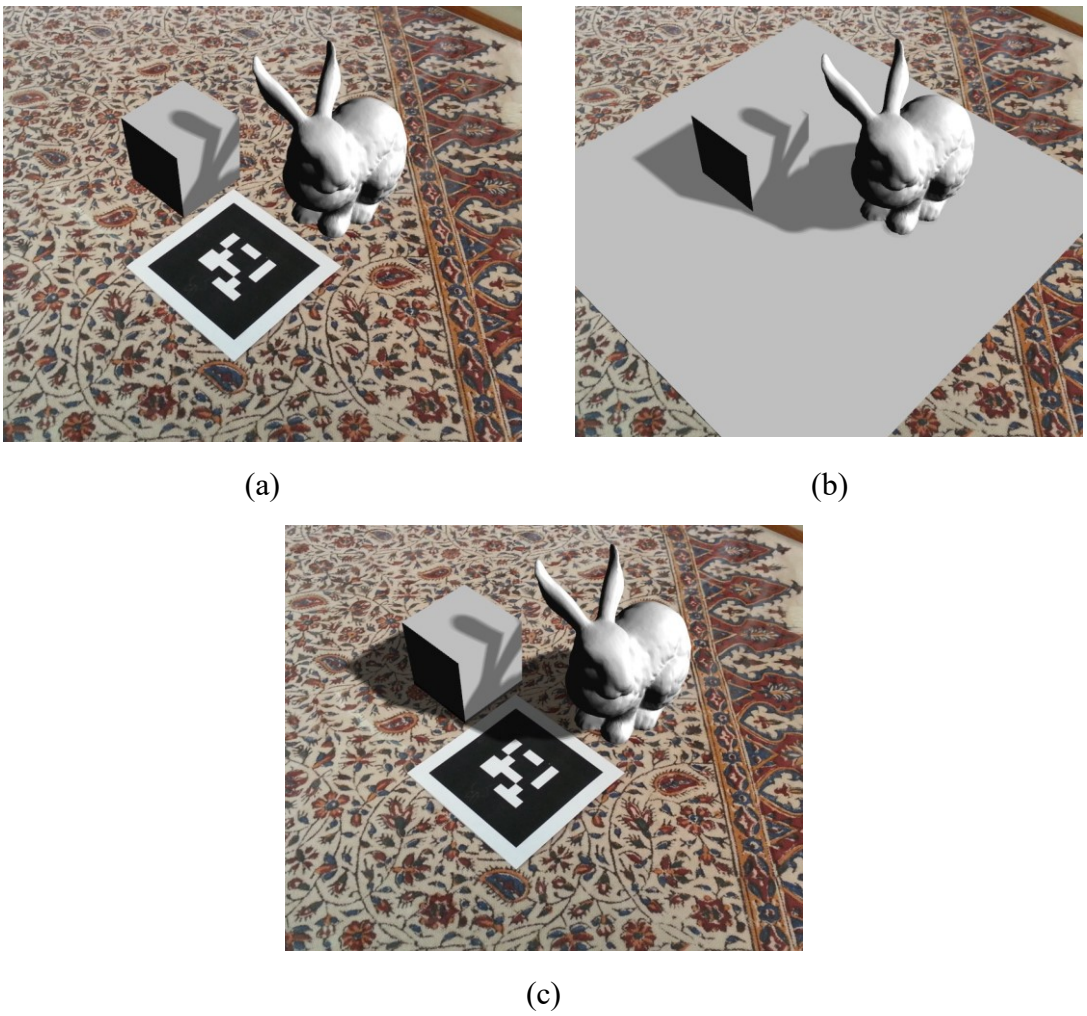
6.1 Desktop AR Applications

The main purpose of the ThirdEye framework is for augmented reality research and development on desktop computer. Three AR demos have been implemented on desktop using this framework so as to demonstrate and evaluate it and show how it can be utilized, and also represent some challenges in AR application development with realistic light effects using rasterizer.

6.1.1 Shadow

In rasterization rendering, usually the shadow volumes or shadow mapping techniques are employed for creating shadow effects, due to their simplicity and performance. In this AR demo, we have used the shadow mapping technique as discussed in Section 3.4 to generate shadow of the virtual 3D object in the real scene.

Generating shadow effect using the shadow mapping technique is straightforward to implement [74], [75]. But in augmented reality environment there is a problem that should be addressed to get more precise result. In the pure virtual world, the generated shadow is cast on another virtual model or surface. In AR environment, there are real objects and surfaces in the scene as well. Casting shadow of the virtual objects on the real object and vice versa is challenging, as the geometries information of the real objects are undefined. The shadow of the virtual model only can be cast on virtual objects or surfaces. In order



(a) (b)

(c)

Figure 32. *Using a virtual plane for casting shadow on real-world content: a) Without plane (Shadow cast only on virtual cube) b) Shadow cast on the visible virtual plane c) Shadow cast on the invisible virtual plane.*

to overcome this problem, the real-world objects can be reconstructed in the virtual environment [29]. These modeled objects are placed in the virtual environment as same position as the real objects in the scene. They are not rendered in output. They are only used for rendering shadow effect.

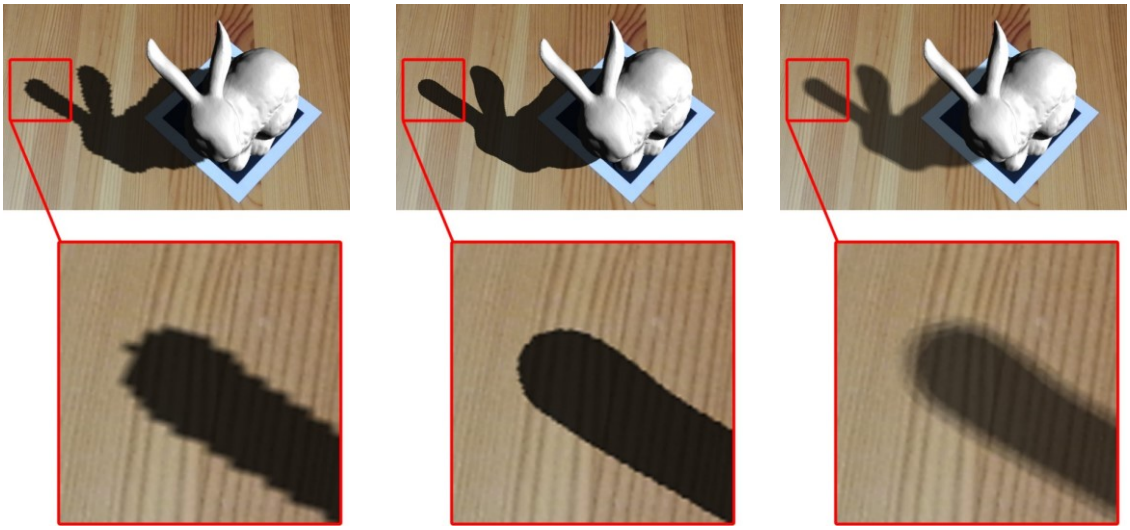
In this demo, the same technique is used for casting shadow of the 3D model on the real table surface. After creating the shadow map, in the second phase a virtual plane that represents the table surface, is used to render the shadow effect. The area of the plane that is not in the shadow should not be rendered. In the fragment shader, each fragment is checked to see whether it is in the shadow area or not. If it is in the shadow it will be shaded otherwise the fragment is discarded. As it is shown in Figure 32(a), the shadow of the bunny is only cast on the virtual cube and it does not appear on the table. Figure 32(b) demonstrates the virtual plane which represents the table surface in the virtual world and the cast shadows are rendered correctly in the scene. The final result is displayed in Figure 32(c), where the rest of plane that are not in the shadow is invisible.

Figure 33. *Exceeded shadow from table surface boundary.*

In the rendering function, first the input image data is rendered as the background image. Then the virtual objects and shadows are rendered onto the background image. The final color for pixels with real-world content that is shadowed by virtual objects is computed based on the alpha channel of the shadow in those pixels. If the shadow alpha channel is zero then the final color for the pixel would be equal to the color of the real-world content. Otherwise, the color of the shadow would be redrawn onto the background which results the final color of the pixel. In this demo the maximum alpha value of the shadow color is less than one. Therefore, the real-world content would be visible in the shadow area.

In this demo the shadow is cast on a flat surface. As it is notable in Figure 33 the shadow of the 3D model is rendered outside the table surface. This is because the virtual plane is larger than surface of the table. Therefore, to produce better result, the real objects in the scene should be modeled accurately and placed in the correct position. Although this technique produces a good result, it has some limitations that make it difficult to be used for all situations. In dynamic scene or a scene with more complex real objects, it would be more challenging to use.

Aliasing is the main problem of the shadow mapping technique [35]. The generated shadow by this technique has jagged edges that has a negative impact on the rendering quality (Figure 34a). One way to overcome this problem is increasing the size of the shadow map [76], [77]. By using a shadow map with larger size, the jaggedness of the shadow border is decreased, but it is still notable (Figure 34b). In order to get better result, we can use blur filtering to soften the shadow border [78] (Figure 34c). For filter, the poison sampling (poison disk) that has been implemented in [74] has been used.



(a) (b) (c)

Figure 34. *Antialiasing shadow: a) Jagged shadow b) Large shadow map (Sharp shadow) c) Blur filtering (Soft shadow).*

6.1.2 Refraction

In the computer graphics, the refraction effect is used to simulate transparent surfaces and objects such as water and glass. In this demo it has been tried to simulate this effect in an augmented reality environment using rasterization rendering.

In order to calculate color of the refractive surface in the specific point, it is required to compute refraction ray at that point. The OpenGL provides *refract* function which can be used in the shader program to calculate refraction ray.

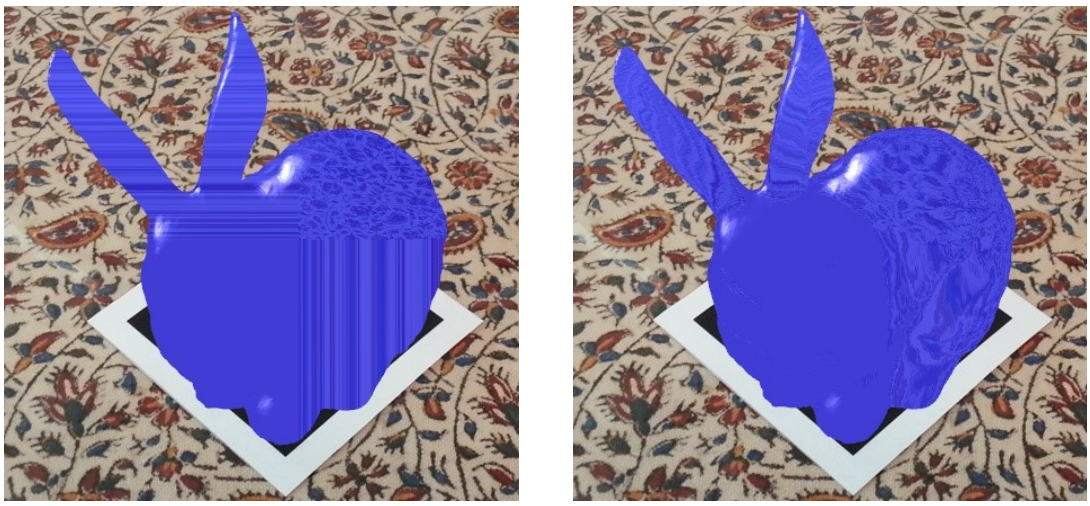
```
vec3 refractionRay = refract(I, N, eta);
```

Where:

- `I` = incident ray (view ray)
- `N` = Surface normal
- `eta` = Ratio of indices of refraction (Refractive index)

After computing refraction ray, the color of the pixel, corresponding to the surface point, can be extracted from a cubemap which contains textures of the surrounding environment. In OpenGL, cubemap is a texture type that contains six individual 2D textures. Generally, it is used for environment mapping.

```
color = texture(cubemapTexture, refractionRay);
```

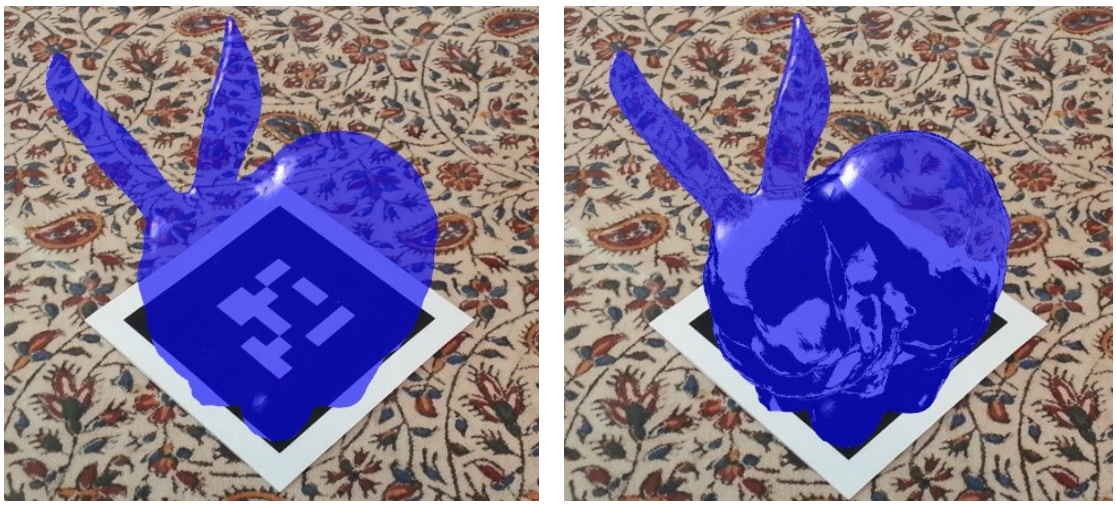


(a) (b)

Figure 35. *Refraction error: a) Refraction index 1.0 b) Refraction index 0.5.*

In a pure virtual environment, creating a cubemap texture and using it for simulating refraction and reflection effect is easy and straightforward. In an augmented reality application, we just have access to an image that comes from the camera. If the refraction ray is used to sample a single texture instead a cubemap, it may yield visual error. Because the ray maybe hit an area outside the texture. This visual error is noticeable in Figure 35. The reflective index of the 3D model in the Figure 35(a) has been set to 1.0 which is reflective index of air medium. So, it should be rendered as a transparent object, however the produced result is not as expected due to erroneous texture sampling.

As it has been mentioned in Section 3.5, one way to overcome this problem is to divide input image to six segments and use them to create an environment cubemap texture. This technique especially useful for simulating reflection light effect. Beside this technique, there is another simpler way to solve this problem and approximately simulate refraction effect in augmented reality environment. The refraction ray can be adjusted in a way that it does not exceed the background image area. As it is shown in Figure 36, this technique can produce a more acceptable result.



(a) (b)

Figure 36. *Refraction correction: a) Refraction index 1.0 b) Refraction index 0.5.*

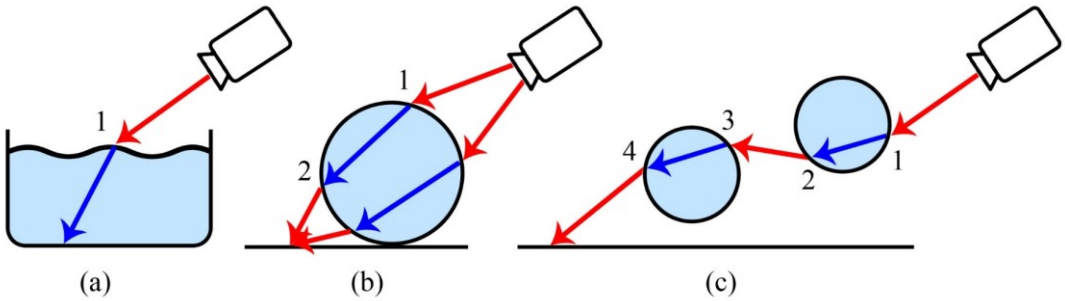


Figure 37. *Refraction ray path with different number of refraction iterations: a) One iteration b) Two iteration c) More than two iterations*

Despite the reasonable achieved result, it is not the same as the real refraction effect. In the real world the ray may refract once or more through the glassy objects before hitting the diffuse surface. For example, a ray refracts just once at the water surface in swimming pool before hitting the bottom of the pool, and it refract twice when go through a glassy ball and so forth (Figure 37). However, the refraction effect with multiple iterations is much complicated and even impossible to calculate using rasterizer. The refraction result in Figure 36 is achieved with only one iteration of refraction.

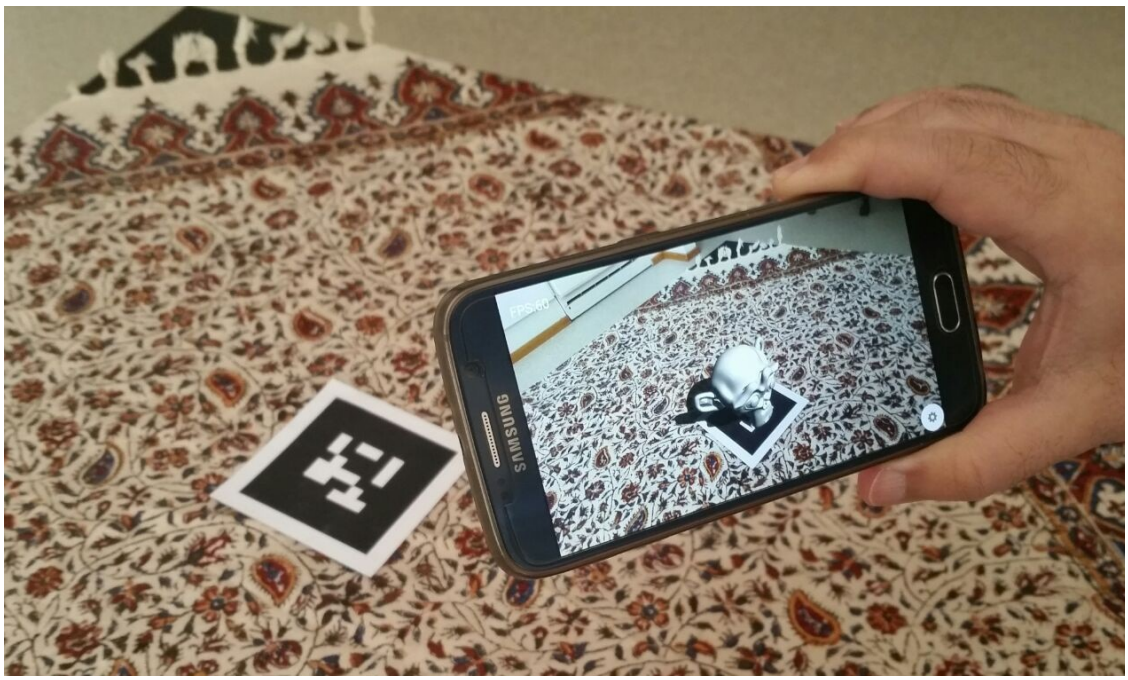


Figure 38. *AR mobile demo using the ThirdEye framework.*

6.2 Mobile AR Applications

In order to demonstrate application of the framework on Android mobile devices, an AR demo is implemented using the ThirdEye framework which simulates shadow effect using the shadow mapping technique, similar to the desktop version demo. In addition, environment light intensity estimation is implemented in the mobile AR demo. Figure 38 shows the mobile AR demo that is implemented using the ThirdEye framework.

The demo is tested on three different mobile devices. After going through the assumptions and introducing the used mobile devices, Section 6.2.1 discusses implemented shadow in mobile version, followed by environment light intensity estimation explanation in Section 6.2.2.

6.2.1 Shadow

In the mobile version of the ThirdEye framework demo, the shadow mapping technique, that has been discussed in Section 6.1.5, was used for creating shadow effect. In contrast to the desktop version, here blur filtering was not applied to the shadow so it has rough border (Figure 39). The size of the shadow map is set to 2048 which reduces aliasing effect. On the other hand, due to the small size display with high *PPI* (pixels per inch), the aliasing effect is less noticeable on the mobile devices.

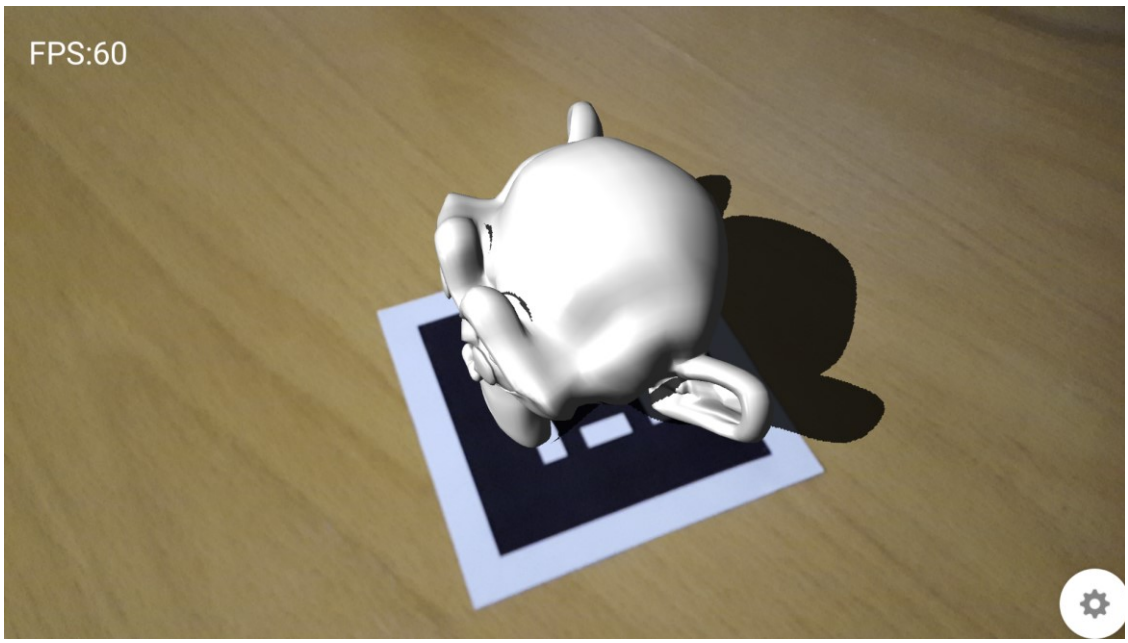


Figure 39. *Shadow – AR mobile demo.*

6.2.2 Environment Light Intensity

Generally, in the augmented reality application a virtual light source is used for rendering virtual models in the scene. Therefore, these virtual models look brighter or darker that they should be respect to the surrounding real-world light intensity. In order to achieve more realistic result, the light intensity of the environment is one of the factor that should be considered during rendering.

In the mobile version of the ThirdEye demo, the average brightness of the current input image is used to adjust the brightness of the virtual model in the scene. As it has been mentioned before, the input image data that is captured by the device camera, is in YUV format. The Y channel in this formant, shows the brightness (Luma) of the image. This makes computing the average brightness easier for us. For each frame, the average of all pixels' brightness is calculated. The result which is a float number is sent to the fragment shader and is used for adjusting brightness of each fragment. If the input image is in the RGB format, the luma of each pixel should calculated first so as to computing the average brightness of the image. Figure 40 demonstrates applying the captured environment light intensity to the rendered virtual object.

(a) (b) (c)

Figure 40. *Environment light intensity estimation:*
a) Before applying b & c) After applying.
a &b) Low light intensity c) High light intensity

6.3 Benchmarking

In order to benchmark the performance of the ThirdEye framework and discover its limitations and bottlenecks of its components, several tests have been conducted. The results of these tests along with some analysis are presented in this section.

6.3.1 Used Hardware and Measurement Setup

The hardware that is used for running and testing the desktop version of the ThirdEye AR demos, is an Asus G55V laptop with following specifications.

- **Processor**: Intel(R) Core(TM)i7-3610QM CPU @ 2.30GHz 2.30GHz
- **GPU**: Nvidia Geforce GTX 660M – 2GB
- **RAM**: 12.0 GB
- **OS**: Linux – Ubuntu 16.04
- **Camera**: Webcam 1.3 Megapixel

The images and videos that has been used for testing the desktop version of the framework were captured by the Samsung Galaxy Note 2.0. In order to run and test the mobile version of the ThirdEye demos, three Android devices with different specifications have been used which are Samsung Galaxy Note 2.0, Samsung Galaxy S6 and Sony Xperia Z Ultra. Table 2 shows specifications of these mobile devices.

Table 2. *Comparison of the used mobile devices' specifications.*

Device	Processor	GPU	Display	Camera	AOS
Note 2.0	Quad-core 1.6 GHz Cortex-A9	Mali-400MP4	5.5" – 16:9 720x1280 PX	8 MP	4.4.2
S6	Octa-core (4x2.1 GHz Cortex-A57 & 4x1.5 GHz Cortex-A53)	Mali-T760MP8	5.1" – 16:9 1440 x 2560 PX	16 MP	7.0
Xperia Z Ultra	Quad-core 2.2 GHz Krait 400	Adreno 330	6.4" – 16:9 1080 x 1920 PX	8 MP	5.1.1

The Sanford bunny model with 86,638 triangles and the Suzanne 3D model with 15,488 triangles were used in desktop and mobile version of the test AR demos, respectively. The power-save mode of the mobile devices was disabled during the tests in order to get more accurate result. In both desktop and mobile versions, the standard chrono library which is part of modern C++ STL, was utilized for measuring the execution time of different operations and calculating FPS in the demos.

6.3.2 Test and Analysis

The AR shadow and refraction demos that were implemented for evaluating usability of the framework (Section 6.1), were used in order to measure performance of the desktop version of the framework as well. The bunny model with 86,638 triangles was used in these demos. The tests were conducted for different light effects for both video and still image with the size of 1920 x 1080 pixel. The result of the tests is presented in Table 3.

Table 3. *Benchmark Results: Rendering different light effects.*

Light effects	Video (FPS)	Image (FPS)
None	57	623
Diffuse + Specular	57	623
Shadow	57	515
Shadow + blur filtering	57	461
Refraction	57	550

It is noticeable from Table 3 that between the demos that use still image as input data, the shadow demo has lowest framerate. The reason is that there are two rendering phases in the shadow demo, as it is mentioned in Section 3.4. Between these AR demos, the diffuse one has highest frame rate as its computationally inexpensive.

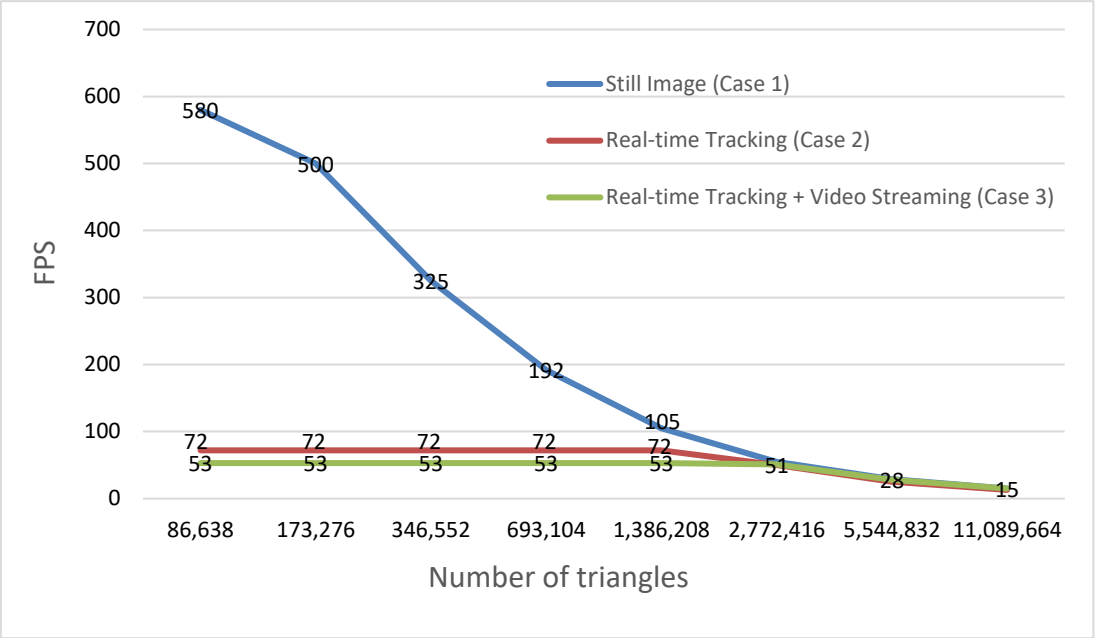


Figure 41. *Benchmark Results: FPS comparison for still image tracking (Case 1), real-time tracking (Case 2) and real-time tracking with video streaming (Case 3), with different number of objects. Image size (1920 x 1080).*

The results in Table 3 show a significant difference between frame rate of the demos that use video and the demos which use still image as input data. All demos with video input, have same frame rates (57 per seconds) although their light effects are different. This is not applied to the AR demos that use still image. For video and camera input, each frame should be captured and sent to the GPU. Updating image data in GPU for each frame is an expensive operation and has a negative impact on rendering performance. In addition, tracking and pose estimation must be performed for each captured image, which also reduces the overall performance.

In order to investigate this issue and examine relation between number of virtual objects in the scene and the frame rate, another test was conducted. In this test the frame rate of the AR demo was measured for different number of objects in three cases. In the first case, an image file data used as input image data. For the second and third cases, the input image data was captured from a video file consecutively. In one of them tracking and pose estimation were done for each frame but the input image data was not sent to GPU. This case acted like augmented reality systems that use optical-see-through display in which the captured image data only is used for tracking and is not displayed. In the third case the tracking was done for each frame and input image data was sent to the GPU. This case performed like video-see-through and monitor-based AR systems. In this test the bunny 3D model was rendered, where diffuse and specular light effects were simulated for every case. The test was executed for the three mentioned cases with different number of objects (1 to 128 bunny models). The results of the test are presented in the Figure 41.

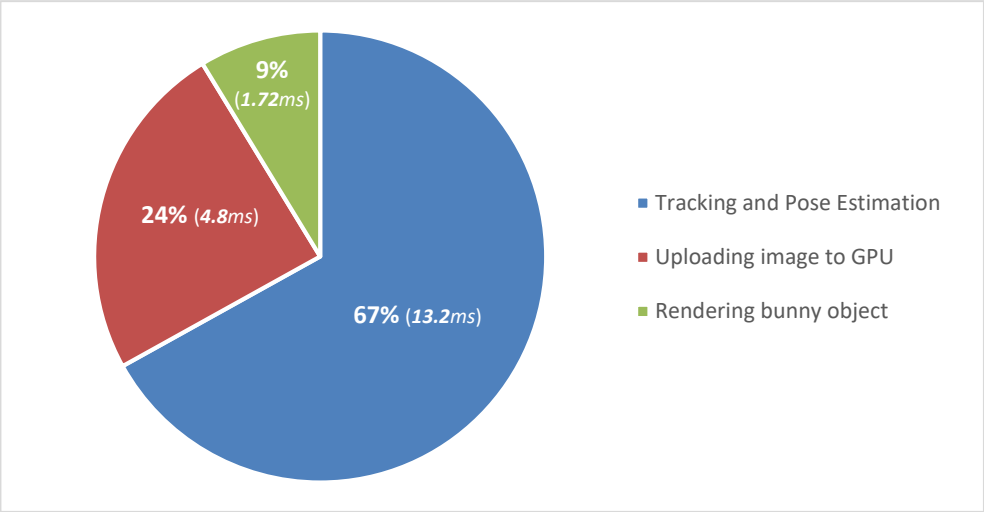


Figure 42. *Benchmark Results: Time taken for one-time Tracking and Pose Estimation and uploading image to GPU and rendering a 3D object.*

As expected, the Figure 41 shows that in the first case, by rendering more virtual objects in the scene, the frame rate is reduced. In this case tracking and uploading the input image data to GPU are done only once. Therefore, they do not affect the rendering performance. However, in the second and third cases, as the Figure 41 shows, the rendering one to sixteen objects in the scene have same effect on the rendering performance of the application. The second case has higher frame rate than the third case, as in the third case besides tracking, image data of each frame should be uploaded to the GPU.

These tests' results show that tracking and uploading the input image data to the GPU memory are main bottlenecks in the framework. From Figure 42 it can be seen that almost 67% of the execution time is consumed by the tracker. Using faster tracking library has a significant impact on overall performance of the framework. Also, we can see that 24% of each frame rendering time is spent on uploading the image to the GPU memory. The optical-see-through AR systems just show augmented content to the users and do not need to display the input image data because the users have direct view of physical world already. As it is shown in Figure 41, we can get better performance if there is no need to display input image data. In the AR demos that were implemented for this thesis work, images with size of 1920 x 1080 pixel were used in order to produce better visual quality. If high quality image is not required by the AR system, using an input image data with lower resolution can result higher tracking and rendering speed.

Table 4 shows benchmark results of tracking and uploading image to GPU tasks' execution time for different image sizes. As it is shown in Table 4, the tracking is performed almost 2.6 times faster for high-definition (HD) image (1280 x 720 pixel) than full HD image (1920 x 1080 pixel). Similarly, the HD image is uploaded to the GPU, 2.7 times

faster than full HD one. In some AR systems, with small size screens, visual quality differences between HD and full HD images are not noticeable. Therefore, using input image data with higher resolution in these kind of systems is pointless and it only causes lower performance of the AR demo application. Although, it should be noted that recognizing the markers which are far from camera in higher resolution input data is easier for the tracker that lower one.

Table 4. *Benchmark Results: Execution time for tracking and uploading image to GPU.*

Image size	Tracking (ms)	Uploading Image to GPU (ms)
1920 x 1080	13.2	4.8
1280 x 720	5.17	1.8
960 x 720	4.2	1.2
640 x 480	1.85	0.48
320 x 240	0.8	0.13

As In order to verify performance improvement, the single thread and multi-thread versions of the color conversion were benchmarked on three mobile devices for two different image sizes. The benchmarking results are displayed in Table 5.

Table 5. *Benchmark Results: Converting YUV to RGB color of an image.*

Device	Image size (960 x 720)		
	Single Thread	Multiple Thread	Improvement %
Note 2.0	150ms	55ms (3 Threads)	172.7%
S6	94ms	44ms (7 Threads)	113.63%
Xperia Z Ultra	160ms	65ms (3 Threads)	146.15%

Device	Image size (1280 x 720)		
	Single Thread	Multiple Thread	Improvement %
Note 2.0	201ms	74ms (3 Threads)	171.62%
S6	117ms	50ms (7 Threads)	134%
Xperia Z Ultra	177ms	80ms (3 Threads)	121.25%

As it is noticeable from Table 5, the multithreaded version of color conversion yields better performance. Although it is expected with more threads the performance is improved subsequently, the benchmark result shows some devices with more threads may have less performance improvement than other devices with less number of threads. For example, in this benchmark Samsung Galaxy S6 with 7 threads had 113.63% to 134% improvement compared to single thread, whereas Samsung Galaxy Note 2.0 with 3 threads had approximately 172% performance enhancement. Usually, there are some applications and services on the mobile phone, running in the background that use CPU resources, therefore all threads may not be available during the conversion process. It might be one of the reasons of the less performance improvement as expected.

As it has been mentioned in Section 5.3.4, a GPU version of the color conversion was implemented using GLSL shader program, which later the CPU version one in the framework was replaced with. In order to verify the performance improvement, the frame rate of CPU and GPU versions were benchmarked. First it should be noted that generally the maximum frame rate that delivered by Android and iOS mobile devices is 60 FPS. For benchmarking, the resolution of the devices' camera was set to 1280 x 720 pixel and the Suzanne model was used for rendering, with shadow and specular effects. The Samsung Note 2 which has less computational power than other two devices, delivers maximum 43 FPS, using CPU version of the color conversion. However, it could render the scene in 58 FPS using GPU version which shows approximately 35% improvement.

The other two devices could achieve approximately 60 FPS in both GPU and CPU version, due to more powerful hardware. Another three benchmarks were executed in order to compare the performances of CPU single thread, multithreaded and GPU versions. For the benchmarking, the Samsung Galaxy S6 that has more powerful hardware than other two devices, was used. In these three benchmarks, 50 Suzanne models (50 x 15,488 triangles) were rendered. The CPU single thread, multithreaded and GPU versions delivered 19, 33 and 40 FPS respectively. The benchmark results show that GPU version of the color conversion results better performance in the framework compared to the other versions.

7. FUTURE WORK

The ThirdEye framework and the applications that utilize this framework can be improved in several ways. The improvement opportunities for future work are listed in this section along with a brief description for each of them.

- ***Light direction estimation:***
  In the AR demos, created for evaluating the framework, the position of the virtual light source has been set manually and it does not represent any of the real light sources present in the scene. In Section 3.7, some of the proposed light estimation methods have been explained. Implementing some of these methods in the ThirdEye framework can help to improve realism aspect of the AR demos that use this framework. GPS approach to estimate sun position is a good candidate method for mobile version of the framework. Most of the mobile devices are equipped with GPS and can be used in outdoor environments.

- ***Environment mapping using input image:***
  Generally, environment mapping technique is used for simulating reflection and refraction light effects, and image-based lighting in computer graphics. There are different techniques that are proposed to create and use environment mapping for different purposes (see Sections 3.5, 3.6 and 3.7). Choosing and implementing a proper environment mapping would be one of the major elements in order to implement light effects in AR.

- ***Reconstructing the physical environment:***
  In augmented reality applications, in order to produce better light effects, such as shadow, and also for more accurate interaction between virtual and real objects in the scene, we need to have better understanding of the real objects' geometries in the physical environment. The RGB-D cameras can be used to scan physical environments. The acquired data can be used to reconstruct real objects in the scene. It would be useful to add support for RGB-D cameras in the ThirdEye framework and provide some utilities to facilitate their use.

- ***Improving the performance in the mobile version of the framework:***
  For the scope of this thesis, it was tried to design and implement the mobile version of the framework in a manner such that it produces acceptable performance on different mobile devices. There are still several areas in this framework that can be optimized further.

- ***Creating photo-realistic AR demo:***
  All the AR demos which have been created for this thesis, use the rasterization rendering technique. It was tried to simulate some of the light effects such as shadow and refraction in AR environment using this technique and in real-time. In order to approximate the physical light effects more accurately, rendering techniques based on ray casting such as path tracing and photon mapping are more suitable and efficient.

- ***Implementing the caustics effect***
  In this work it was tried to implement caustics in an augmented reality environment using a rasterizer. However, the progress of the implementation was terminated due to the thesis time limitation. The techniques that were experimented on during this work were mentioned in Section 3.7, which might be helpful for the future development.

- ***Using IMU and GPS for hybrid tracking:***
  IMU and GPS sensors in the mobile devices can be used for hybrid tracking along with already existing marker-based tracking in the framework.

- ***Adding user interaction functionality:***
  Interacting with the virtual objects in the augmented environment gives more natural feelings to the user. In order to simplify the development of an AR demo with user interaction feature, some functionalities can be added to the framework which facilitate handling the user interaction.

8. CONCLUSIONS

Generally, augmented reality systems consist of multiple components. These components are common between every AR system. However, a system may use different technologies for each component compared to other AR system. The ThirdEye framework was designed and implemented based on the common structure of augmented reality systems, in order to prevent redundancy and to facilitate AR application development.

This thesis investigated augmented reality systems and described their structures and main components. A research was done about the existing toolkits and SDKs for implementing AR applications. Some criteria were considered in order to choose proper tools and libraries that are utilized in the framework. As it was desired to port this framework to other platforms easily, the selected tools should be cross-platform and open-source.

The ThirdEye framework was developed using object oriented programming paradigm in order to achieve maintainability, reusability, and convenient software design. The framework was developed for desktop and Android mobile platforms. It was tried to keep consistency between both versions so as to minimize the required modifications. An AR demo application using ThirdEye can be ported to both supported platforms with ease, compared to the various existing libraries. The ThirdEye framework includes useful features for AR demo application development for experimental purposes. These features may be provided by the other existing frameworks separately, however, this framework provides those features in an easy-to-use manner.

In this thesis the structure of both versions of the ThirdEye framework were explained and compared. The instructions for using the framework was described with a simple AR demo. The implementation of each demo and the encountered challenges were described in detail. In order to evaluate the usability of the ThirdEye framework, several AR demos were developed using the framework. Shadow and refraction light effects in AR environment were implemented in these demos. In addition, light intensity estimation was applied to the mobile AR demo.

Furthermore, the implemented AR demo applications were benchmarked so as to investigate the bottlenecks. Based on the benchmarks' results, tracking and uploading the input image data to GPU were the most time-consuming processes, compared to the rendering the virtual objects. Size of the input image data has significant impact on the overall performance and it should be set with proper size in order to obtain reasonable results with better performance. Besides, the mobile AR demo was tested on three different mobile phones and the single and multi-threaded versions of the color conversion were benchmarked.

Finally, several ideas were presented for future improvement of the ThirdEye framework. For example, estimating light direction by using GPS data, adding support for RGB-D cameras so as to reconstruct the physical environment, and adding user interaction functionality to the framework in order to improve the usability of the ThirdEye framework.

REFERENCES

[1] J. Carmigniani, B. Furht, M. Anisetti, P. Ceravolo, E. Damiani, and M. Ivkovic, "Augmented reality technologies, systems and applications," *Multimed. Tools Appl*, vol. 51, no. 1, pp. 341–377, 2011.

[2] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre, "Recent advances in augmented reality," *IEEE Comput. Graph. Appl.*, vol. 21, no. 6, pp. 34–47, Dec. 2001.

[3] T. P. Caudell and D. W. Mizell, "Augmented reality: an application of heads-up display technology to manual manufacturing processes," 1992, pp. 659–669 vol.2.

[4] P. Milgram and F. Kishino, *A Taxonomy of Mixed Reality Visual Displays*, vol. E77-D, no. 12. 1994.

[5] H. Regenbrecht *et al.*, "Using Augmented Virtuality for Remote Collaboration," *Presence Teleoperators Virtual Environ.*, vol. 13, no. 3, pp. 338–354, Jun. 2004.

[6] I. E. Sutherland, "A Head-mounted Three Dimensional Display," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, New York, NY, USA, 1968, pp. 757–764.

[7] B. H. Thomas and C. Sandor, "What Wearable Augmented Reality Can Do for You," *IEEE Pervasive Comput.*, vol. 8, no. 2, pp. 8–11, Apr. 2009.

[8] H. Lopez, A. Navarro, and J. Relano, "An Analysis of Augmented Reality Systems," 2010, pp. 245–250.

[9] R. T. Azuma, "A Survey of Augmented Reality," *Presence Teleoperators Virtual Environ.*, vol. 6, no. 4, pp. 355–385, Aug. 1997.

[10] M. Mihelj, D. Novak, and S. Begus, "Augmented Reality," in *Virtual Reality Technology and Applications*, Springer, Dordrecht, 2014, pp. 195–204.

[11] M. Pharr and G. Humphreys, *Physically based rendering: from theory to implementation*, 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2010.

[12] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 11, pp. 1330–1334, Nov. 2000.

[13] J. Rekimoto, "Matrix: a realtime object identification and registration method for augmented reality," 1998, pp. 63–68.

[14] D. Wagner and D. Schmalstieg, "ARToolKitPlus for Pose Tracking on Mobile Devices," 2007.

[15] F. Bergamasco, A. Albarelli, and A. Torsello, "Image-Space Marker Detection and Recognition Using Projective Invariants," 2011, pp. 381–388.

[16] J. Barandiaran and D. Borro, "Edge-Based Markerless 3D Tracking of Rigid Objects," in *17th International Conference on Artificial Reality and Telexistence (ICAT 2007)*, 2007, pp. 282–283.

[17] T. Lee and T. Hollerer, "Handy AR: Markerless Inspection of Augmented Reality Objects Using Fingertip Tracking," 2007, pp. 1–8.

[18] R. A. Newcombe *et al.*, “KinectFusion: Real-time dense surface mapping and tracking,” in *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, 2011, pp. 127–136.

[19] R. R. Brooks and S. S. Iyengar, *Multi-sensor Fusion: Fundamentals and Applications with Software*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[20] M. Koskela, “Software-Based Ray Tracing for Mobile Devices,” Aug. 2015.

[21] A. Appel, “Some Techniques for Shading Machine Renderings of Solids,” in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, New York, NY, USA, 1968, pp. 37–45.

[22] T. Whitted, “An Improved Illumination Model for Shaded Display,” in *ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005.

[23] H. W. Jensen, *Realistic image synthesis using photon mapping*. Natick, MA: A K Peters, 2001.

[24] J. T. Kajiya, “The Rendering Equation,” in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1986, pp. 143–150.

[25] R. L. Cook, T. Porter, and L. Carpenter, “Distributed Ray Tracing,” in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1984, pp. 137–145.

[26] H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping*. Natick, MA, USA: A. K. Peters, Ltd., 2001.

[27] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, F. Sillion, and A. Gravir/Imag-Inria, “A Survey of Real-time Soft Shadows Algorithms,” 2003.

[28] Z. Noh and M. S. Sunar, “A Review of Shadow Techniques in Augmented Reality,” in *Machine Vision, International Conference on*, Los Alamitos, CA, USA, 2009, pp. 320–324.

[29] M. Haller, S. Drab, and W. Hartmann, “A Real-time Shadow Approach for an Augmented Reality Application Using Shadow Volumes,” in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, New York, NY, USA, 2003, pp. 56–65.

[30] A. Fournier, A. S. Gunawan, and C. Romanzin, “Common Illumination Between Real and Computer Generated Scenes,” University of British Columbia, Vancouver, BC, Canada, Canada, 1992.

[31] S. Pessoa, G. Moura, J. Lima, V. Teichrieb, and J. Kelner, “Photorealistic rendering for Augmented Reality: A global illumination and BRDF solution,” in *2010 IEEE Virtual Reality Conference (VR)*, 2010, pp. 3–10.

[32] A. L. D. Santos, D. Lemos, J. E. F. Lindoso, and V. Teichrieb, “Real Time Ray Tracing for Augmented Reality,” in *2012 14th Symposium on Virtual and Augmented Reality*, 2012, pp. 131–140.

[33] D. Eigen, C. Puhrsch, and R. Fergus, “Depth Map Prediction from a Single Image using a Multi-Scale Deep Network,” in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2366–2374.

[34] F. C. Crow, “Shadow algorithms for computer graphics,” in *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, 1977, pp. 242–248.

[35] S. Brabec, T. Annen, and H.-P. Seidel, “Practical Shadow Mapping,” *J. Graph. Tools*, vol. 7, no. 4, pp. 9–18, Jan. 2002.

[36] T. Ropinski, S. Wachenfeld, and K. Hinrichs, “Virtual reflections for augmented reality environments,” in *Int. Conference on Artificial Reality and Telexistence*, 2004, pp. 311–318.

[37] M. A. Shah, J. Konttinen, and S. Pattanaik, “Caustics Mapping: An Image-Space Technique for Real-Time Caustics,” *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 2, pp. 272–280, Mar. 2007.

[38] K. Agusanto, L. Li, Z. Chuangui, and N. W. Sing, “Photorealistic rendering for augmented reality using environment illumination,” in *The Second IEEE and ACM International Symposium on Mixed and Augmented Reality, 2003. Proceedings.*, 2003, pp. 208–216.

[39] P. Kán and H. Kaufmann, “High-quality reflections, refractions, and caustics in Augmented Reality and their contribution to visual coherence,” in *2012 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2012, pp. 99–108.

[40] H. Croubois, J.-P. Farrugia, and J.-C. Iehl, “Fast Image Based Lighting for Mobile Realistic AR,” LIRIS UMR CNRS 5205 ; ENS Lyon, Research Report, 2014.

[41] K. Rohmer, J. Jendersie, and T. Grosch, “Natural Environment Illumination: Coherent Interactive Augmented Reality for Mobile and Non-Mobile Devices,” *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 11, pp. 2474–2484, Nov. 2017.

[42] T. Saito and T. Takahashi, “Comprehensible Rendering of 3-D Shapes,” in *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1990, pp. 197–206.

[43] R. Marroquim, M. Kraus, and P. R. Cavalcanti, “Efficient Point-Based Rendering Using Image Reconstruction.,” in *SPBG*, 2007, pp. 101–108.

[44] M. Colbert and J. Křivánek, “GPU-Based Importance Sampling,” in *GPU Gems 3*, Addison-Wesley Professional, 2007.

[45] L. Szirmay-Kalos, B. Aszódi, I. Lazányi, and M. Premecz, “Approximate Ray-Tracing on the GPU with Distance Impostors,” *Comput. Graph. Forum*, vol. 24, no. 3, pp. 695–704, Sep. 2005.

[46] E. Wallace, “Rendering Realtime Caustics in WebGL,” *Evan Wallace*, 07-Jan-2016. .

[47] M. McGuire and D. Luebke, “Hardware-accelerated Global Illumination by Image Space Photon Mapping,” in *Proceedings of the Conference on High Performance Graphics 2009*, New York, NY, USA, 2009, pp. 77–89.

[48] S. Reznik, “Modeling of glass surfaces.” [Online]. Available: http://www.uraldev.ru/articles/id/39/page/1.

[49] T. Richter-Trummer, D. Kalkofen, J. Park, and D. Schmalstieg, “Instant Mixed Reality Lighting from Casual Scanning,” in *2016 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2016, pp. 27–36.

[50] P. E. Debevec and J. Malik, “Recovering High Dynamic Range Radiance Maps from Photographs,” in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1997, pp. 369–378.

[51] P. Debevec, “Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-based Graphics with Global Illumination and High Dynamic Range Photography,” in *ACM SIGGRAPH 2008 Classes*, New York, NY, USA, 2008, pp. 32:1–32:10.

[52] M. Meilland, C. Barat, and A. Comport, “3D High Dynamic Range dense visual SLAM and its application to real-time object re-lighting,” in *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2013, pp. 143–152.

[53] M. Knecht, C. Traxler, O. Mattausch, W. Purgathofer, and M. Wimmer, “Differential Instant Radiosity for mixed reality,” in *2010 IEEE International Symposium on Mixed and Augmented Reality*, 2010, pp. 99–107.

[54] Jan-Michael Frahm, Kevin Koeser, Daniel Grest, and Reinhard Koch, “Markerless Augmented Reality with Light Source Estimation for Direct Illumination,” presented at the The 2nd IEE European Con- ference, 2005, pp. 211–220.

[55] P. Lagger and P. Fua, “Using Specularities to Recover Multiple Light Sources in the Presence of Texture,” in *18th International Conference on Pattern Recognition (ICPR’06)*, 2006, vol. 1, pp. 587–590.

[56] T. Mashita, H. Yasuhara, A. Plopski, K. Kiyokawa, and H. Takemura, “In-situ lighting and reflectance estimations for indoor AR systems,” in *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2013, pp. 275–276.

[57] J. Jachnik, R. A. Newcombe, and A. J. Davison, “Real-time surface light-field capture for augmentation of planar specular surfaces,” in *Mixed and Augmented Reality (ISMAR), 2012 IEEE International Symposium on*, 2012, pp. 91–97.

[58] L. Gruber, T. Richter-Trummer, and D. Schmalstieg, “Real-time photometric registration from arbitrary geometry,” in *2012 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2012, pp. 119–128.

[59] R. Ramamoorthi and P. Hanrahan, “A Signal-processing Framework for Inverse Rendering,” in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 2001, pp. 117–128.

[60] B. J. Boom, S. Orts-Escolano, X. X. Ning, S. McDonagh, P. Sandilands, and R. B. Fisher, “Interactive light source position estimation for augmented reality with an RGB-D camera,” *Comput. Animat. Virtual Worlds*, vol. 28, no. 1, p. n/a-n/a, Jan. 2017.

[61] Y. Uranishi, M. Imura, and T. Kuroda, “The Rainbow Marker: An AR marker with planar light probe based on structural color pattern matching,” in *2016 IEEE Virtual Reality (VR)*, 2016, pp. 303–304.

[62] C. B. Madsen and M. Nielsen, “Towards probe-less augmented reality,” *Position Pap. Comput. Vis. Media Technol. Lab Aalb. Univ. Aalb. Den.*, 2008.

[63] Madsen and Lal, “Probeless Illumination Estimation for Outdoor Augmented Reality,” 2010.

[64] T. K. d Castro, L. H. d Figueiredo, and L. Velho, “Realistic Shadows for Mobile Augmented Reality,” in *2012 14th Symposium on Virtual and Augmented Reality*, 2012, pp. 36–45.

[65] H. Kato and M. Billinghurst, “Marker tracking and HMD calibration for a video-based augmented reality conferencing system,” 1999, pp. 85–94.

[66] *artoolkit-docs: Documentation files (.md) for ARToolKit*. ARToolKit, 2018.

[67] S. Siltanen and Valtion teknillinen tutkimuskeskus, *Theory and applications of marker-based augmented reality*. 2012.

[68] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, “Automatic generation and detection of highly reliable fiducial markers under occlusion,” *Pattern Recognit.*, vol. 47, no. 6, pp. 2280–2292, Jun. 2014.

[69] “Fundamental Concepts | ARCore,” *Google Developers*. [Online]. Available: https://developers.google.com/ar/discover/concepts.

[70] “ARCore - Virtual Reality and Augmented Reality Wiki - VR & AR Wiki.” [Online]. Available: https://xinreality.com/wiki/ARCore.

[71] “About Augmented Reality and ARKit | Apple Developer Documentation.” [Online]. Available: https://developer.apple.com/documentation/arkit/about_augmented_reality_and_arkit#see-also.

[72] “Camera calibration With OpenCV — OpenCV 2.4.13.6 documentation.” [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html.

[73] M. Haghighat, “Read YUV Videos and Extract the Frames - File Exchange - MATLAB Central.” [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/59497-read-yuv-videos-and-extract-the-frames.

[74] “Tutorial 16 : Shadow mapping.” [Online]. Available: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/.

[75] “LearnOpenGL - Shadow Mapping.” [Online]. Available: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping.

[76] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg, “Adaptive Shadow Maps,” in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 2001, pp. 387–390.

[77] M. Stamminger and G. Drettakis, “Perspective Shadow Maps,” in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 2002, pp. 557–562.

[78] T. Annen, T. Mertens, P. Bekaert, H.-P. Seidel, and J. Kautz, “Convolution Shadow Maps,” in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 51–60.